LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

AD-A209 118

MIT/LCS/TR-439

VERIFYING OBJECT-ORIENTED PROGRAMS THAT USE SUBTYPES

Gary Todd Leavens

February 1989

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY 2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
4. PERFORMING ORGANIZATION REPORT NUMBER(S) MIT/LCS/TR-439	5. MONITORING ORGANIZATION REPORT NUMBER(S) N00014-83-K-0125
6a. NAME OF PERFORMING ORGANIZATION MIT Laboratory for Computer Science 6b. OFFICE SYMBOL *(If applicable)*	7a. NAME OF MONITORING ORGANIZATION Office of Naval Research/Department of Navy
6c. ADDRESS *(City, State, and ZIP Code)* 545 Technology Square Cambridge, MA 02139	7b. ADDRESS *(City, State, and ZIP Code)* Information Systems Program Arlington, VA 22217
8a. NAME OF FUNDING / SPONSORING ORGANIZATION DARPA/DOD 8b. OFFICE SYMBOL *(If applicable)*	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS *(City, State, and ZIP Code)* 1400 Wilson Blvd. Arlington, VA 22217	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE *(Include Security Classification)*
Verifying Object-Oriented Programs that use Subtypes

12. PERSONAL AUTHOR(S)
Leavens, Gary Todd

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT *(Year, Month, Day)*	15. PAGE COUNT
Technical	FROM ______ TO ______	1989 February	208

16. SUPPLEMENTARY NOTATION
[illegible]

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
programming languages, object-oriented, Smalltalk, specification, subtype, type checking, abstract type, generic invocation, message passing, inclusion polymorphism. (KT)

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

Object-oriented programming languages like Smalltalk-80 have a generic invocation mechanism that allows code to work on instances of many different types. In this dissertation we show how to write formal specifications of functions that use generic invocation and give a logic for verifying applicative programs that use generic invocation.

Our reasoning techniques formalize informal methods based on the use of subtypes. We give a formal definition of subtype relationships among immutable abstract types, including nondeterministic and incompletely specified types. This definition captures the intuition that each instance of a subtype behaves like some instance of that type's supertypes. We show how to write specifications of functions that use generic invocation by allowing instances of subtypes as arguments. We also simplify verification by separately checking that each expression's value is an instance of a subtype of the expression's type.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Judy Little, Publications Coordinator	22b. TELEPHONE *(Include Area Code)* (617) 253-5894 22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

U.S. Government Printing Office: 1985—507-047

Unclassified

89 6 15 043

Verifying Object-Oriented Programs that use Subtypes

by

Gary Todd Leavens

December 1988

This work was supported by the Defense Advanced Research Projects Agency (DARPA) under Grant N00014-83-K-0125.

Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, Massachusetts 02139

To Janet,
who brought the spring that ended my coldest winter,
and whose summer sunshine adds happiness to my life.

Verifying Object-Oriented Programs that use Subtypes

by

Gary Todd Leavens

Submitted to the Department of
Electrical Engineering and Computer Science on December 21, 1988
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Computer Science

Abstract

Object-oriented programming languages like Smalltalk-80 have a generic invocation mechanism that allows code to work on instances of many different types. In this dissertation we show how to write formal specifications of functions that use generic invocation and give a logic for verifying applicative programs that use generic invocation.

Our reasoning techniques formalize informal methods based on the use of subtypes. We give a formal definition of subtype relationships among immutable abstract types, including nondeterministic and incompletely specified types. This definition captures the intuition that each instance of a subtype behaves like some instance of that type's supertypes. We show how to write specifications of functions that use generic invocation by allowing instances of subtypes as arguments. We also simplify verification by separately checking that each expression's value is an instance of a subtype of the expression's type.

Thesis Supervisor: William E. Weihl
Title: Associate Professor of Computer Science

Keywords: programming languages, object-oriented, Smalltalk, specification, verification, subtype, type checking, abstract type, generic invocation, message passing, inclusion polymorphism.

1987 CR Categories: D.2.1 [*Software Engineering*] Requirements/Specifications — Languages; D.2.4 [*Software Engineering*] Program Verification – Correctness proofs; D.3.3 [*Programming Languages*] Language Constructs — Abstract data types, procedures, functions, and subroutines; F.3.1 [*Logics and Meanings of Programs*] Specifying and verifying and reasoning about programs — logics of programs, pre- and post-conditions, specification techniques.

List of Figures

Contents

Biographical Note

Gary Leavens was born in Royal Oak, Michigan in November 1957 to Neil and Burnis Leavens. He had happy childhood in Royal Oak, where he was graduated from Kimball High School in June, 1975. He attended the University of Michigan, in Ann Arbor, where he received a Bachelor of Science degree in December 1978, with High Distinction and Honors in Computer and Communication Sciences. Having finished a term early, he enrolled in the graduate school at the University of Michigan, where he stayed for one more term.

In May 1979 Gary started work as a member of technical staff for Bell Telephone Laboratories in Denver, Colorado. At Bell Laboratories he worked on the development and maintenance of a large software management system. As part of the One Year on Campus program of Bell Laboratories, he attended the University of Southern California in Los Angeles, California from September 1979 until June 1980, when he received a Master of Science degree in Computer Science.

In September 1982 Gary enrolled at the Massachusetts Institute of Technology. At MIT he received a GenRad/AEA faculty development fellowship. He was a teaching assistant for several graduate and undergraduate courses and was active in the design and implementation of the Argus distributed programming language and system. He is a coauthor and the editor of the *Argus Reference Manual.*

Starting in January 1989, Gary will be an assistant professor of computer science at the Iowa State University of Science and Technology, in Ames, Iowa.

Acknowledgements

My deepest thanks go to Bill Weihl, who not only helped me develop my ideas, but who is also an excellent reader and critic, and a good friend. Barbara Liskov was also instrumental in developing and organizing my ideas and she encouraged me to pursue the topic of subtyping. Barbara also provided much of my financial support during graduate school, and she always shows me that she believes in me and my work. John Guttag provided technical expertise, encouragement, and a dedication to clear writing and thinking. These three formed my dissertation committee.

Many others also contributed to the development of my ideas. Toby Bloom helped me develop my topic and was involved in many discussions. Bob Scheifler was always excellent in his constructive criticisms. Craig Schaffert provided critical comments, examples, and many discussions. Alan Snyder helped point out the importance of incomplete specifications. Jeannette Wing helped by steering me to model theory. Dan Carnese and Rich Zippel helped me understand existing object-oriented systems and programs. My primary reference for nondeterministic algebras is an article by Tobias Nipkow, whom I have had the good fortune to talk with in person during this past year. John Mitchell took time for technical discussions, and his work on representation independence was an inspiration for me. Others who helped by taking time for technical discussions and by asking good questions were: Dave Gifford, Kathy Yelick, Suresh Jagannathan, Rishiyur Nikhil, Val Breazu-Tannen, Albert Meyer, Jonathan Rees, Earl Waldin, Jon Riecke, Lalita Jategaonkar, John Lucassen, Mark Reinhold, Brian Oki, Kim Bruce, and Peter Wegner. There were also several professors at schools where I interviewed with whom I had stimulating technical discussions, especially Laurette Bradley (at UCSD), Al Baker and Jim Bieman (at Iowa State), and Chris Haynes and Dan Friedman (at Indiana). To these that I have named, to all my teachers, and to those I have forgotten, my heartfelt thanks.

To God and to all my friends and family, my thanks for more than I can say. My parents and family have been a constant source of love and encouragement. In more recent months my loving wife, Janet, has given me great joy and has supported me in every way. My family and my friends, Brian Oki, Dane Krampitz, John Morrison, Bill Weihl, Susan and Michael Thomas, Rachael Rassmussen, Fred Reisz, Janet Shaffer, Paul Hauge, Angella Carmella, Brian and Dinah Dalder, and Andrew Xu were all instrumental in helping me through the most difficult time of my life; they are truly friends indeed. I am proud and happy to count many others as my friends, although I have not leaned on them so heavily; these include Kathy Yelick, Minoru Kubota, Debbie Hwang, Elliot Kolodner, Suresh Jagannathan, Robert Scheifler, Jon Riecke, Paul Hillner, David Jaeger, Sharon Perl, Mark Day, Paul Johnson, Dorothy Curtis, Toby Bloom, Boaz Ben-Zvi, Sanjay Ghemawat, Robert Gruber, Rivka Ladin, Barbara Liskov, John Guttag, Ann Rubin, Liuba Shrira, Daniel Jackson, Tobias Nipkow, Arch Oldehoeft, Nancy Lynch, Albert Meyer, Jennifier Welsh, Mike Vermeulen, Ed Walker, Jim Restivo, Randy Forgaard, Maurice Herlihy, Jeannette Wing, Sheng-Yang Chiu, David Beyer, Dick Schoenfeldt, Radia Pearlman, Tony and Arelene Graves, Dan and Carole Sendek, Paul Borucki, and many others at MIT that I do not have space to name.

I am grateful for the financial support of the MIT Laboratory for Computer Science (through Professor Liskov) and for the GenRad Foundation for funding my GenRad/AEA faculty development fellowship. Thanks also to Virginia Boynton for her support until December 1987.

Chapter 1

Introduction

Object-oriented programming languages, such as Smalltalk-80 [GR83], provide programmers with powerful tools. One of these tools is a generic invocation (or message passing) mechanism, which allows code to manipulate instances of different types. To reason about programs that use generic invocation, programmers often classify types by how instances of that type behave. If each instance of type S behaves like an instance of type T, then S is called a subtype of T. Programmers hope that if their code works correctly when it operates on instances of some type T, then their code will also work correctly on instances of subtypes of T. However, since they lack the formal tools to guide their classification of types and their reasoning, their reasoning lacks certainty.

In this dissertation, we take the following steps towards a foundation for reasoning about programs that use generic invocation.

- We present a formal definition of subtyping among abstract types.

- We present formal techniques for specifying and verifying programs that use generic invocation.

1.1 Generic Invocation

To explain generic invocation, we must first define a few terms. An *abstract data type* is a description of a set of *instances* (or objects) and a set of operations that create and manipulate them [LG86]. A *class* is a program module that implements an abstract data type. A class defines a representation for instances and implements the various operations of the abstract type with procedures. Each abstract data type has a name, and we give a class the same name as the abstract type it implements. The names of abstract types are the *types* used in type checking. Each object is an instance of the class that created it, and thus an instance of the abstract type that the class implements. Therefore each instance has one and only one class and type, and its class and type have the same name. For example, an instance of type `Text_Window` would also be an instance of the class named `Text_Window`.

The invocations of `read_line(io)` in the second and last lines of the Trellis/Owl [SCB*86] [OBr85] [SCW85] program of Figure 1.1 are *generic invocations*, because they are syntactically identical and yet may result in the running of different procedures. In the second line of the program, `io` denotes an instance of `Text_Window`, so the generic invocation mechanism runs code that is part of the class `Text_Window`. In the last line of the program, code from the class `IO_Stream` will be run if `io` denotes an instance of `IO_Stream`, and otherwise the code from the class `Text_Window` will be run. Notice that in the last line, the choice of what code to invoke cannot be made before the result of the `if` expression is known.

Generic invocation is sometimes called *message passing* because one can think of `read_line(io)` as sending the message "`read_line`" to the object denoted by `io`. In this metaphor, a message is an operation symbol together with the remaining arguments.

Generic invocation can be thought of as a dynamic form of overloading, but it should not be confused with static (i.e., compile-time) overloading, as found, for example, in Ada [Ada83]. In an Ada program, both `IO_Stream` and `Text_Window` can have operations named `read_line`. However, the procedure that is called by `read_line(io)` in Ada depends statically on the type of the *variable* `io` instead of depending dynamically on the type of the *object* bound to `io`. Since in Ada the same variable cannot refer to

Figure 1.1: Example of generic invocation.

```
var io:  IO_Stream := create(Text_Window, ...);
ans:  String := read_line(io);
io := if (ans = "y")
      then create(Text_Window, ...)
      else create(IO_Stream, ...)
      end if;
l:  String := read_line(io)
```

instances of different types, syntactically different invocations would be needed to call each of the two `read_line` operations.

Trellis/Owl, like Smalltalk-80, treats all invocations of operations that occur in a program as generic invocations. An invocation such as `create(Text_Window,...)` is a generic invocation that takes the `Text_Window` class object and some other arguments and returns an instance of `Text_Window`. (A *class object* is an object that represents a type at run-time.) Since `create` takes a class object as an argument, it is a *class operation*. Operations that do not take class objects as arguments, such as `read_line`, are called *instance operations*.

1.2 Inclusion Polymorphism

By using generic invocation, one can write *polymorphic* procedures; that is, procedures that can be applied to instances of different types. For example, the Trellis/Owl procedure `sumFirst` in Figure 1.2 is polymorphic. This polymorphism is exhibited by the call,

```
sumFirst(make(IntPair,1,2), make(IntTriple,4,5,6)),
```

whose result is 5. `IntPair` is a type with a class operation `make` and instance operations `first` and `second`. The type `IntTriple` also has an instance operation named `first`, as well as a class operation `make` and instance operations `second` and `third`. Since we can pass to `sumFirst` instances of either `IntPair` or `IntTriple`, `sumFirst` is polymorphic.

Cardelli and Wegner have coined the term *inclusion polymorphism* for the kind of polymorphism exhibited by `sumFirst` [CW85, Page 475]. Unlike other kinds of polymorphism, inclusion polymorphism implicitly associates a set of named instance operations with each object. For example, Trellis/Owl implicitly associates the instance operations `first` and `second` with each instance of `IntPair`.

Figure 1.2: Implementation of the function `sumFirst` in Trellis/Owl.

```
proc sumFirst(p1,p2: IntPair) returns(Int)
    return(add(first(p1),first(p2)))
end;
```

By contrast, consider writing `sumFirst` in a language with *parametric polymorphism*, such as ML [GMW79]. In ML we would have to explicitly pass the operations that extract the first components of the objects, because an object's instance operations are not available at run-time in ML. (See Figure 1.3.) However, the ML version of `sumFirst` is more general than the Trellis/Owl version, because the first two arguments do not have to be instances of types that have operations named `first`.

Inclusion polymorphism has two main advantages.

First, programs may be more terse than in a language with parametric polymorphism, since instance operations are implicitly associated with objects, and thus extra parameters for types or operations need not be mentioned.

Second, the type of a polymorphic procedure mentions the names of existing types. For example, the declared type of the Trellis/Owl version of `sumFirst` is

$$\texttt{IntPair}, \texttt{IntPair} \rightarrow \texttt{Int},$$

which means that all types that are subtypes of `IntPair` (such as `IntTriple`) can be used as arguments. The type of the ML version of `sumFirst`, which is

$$\texttt{T1}, \texttt{T2}, (\texttt{T1} \rightarrow \texttt{Int}), (\texttt{T2} \rightarrow \texttt{Int}) \rightarrow \texttt{Int}$$

for all types `T1` and `T2`, does not mention existing types such as `IntPair`. One advantage of mentioning existing types is that the type system can check that each argument has a type that is a subtype of the corresponding formal argument type. We will show later how this property can be used in program verification. In ML, the type system does not have such semantic information about types and cannot aid one's verification to the same extent as in Trellis/Owl.

Figure 1.3: Implementation of the function `sumFirst` in ML.

```
val sumFirst(p1,p2,first1,first2) =
    first1(p1) + first2(p2)
```

1.3 Specification and Verification Problems

In this section we describe the problem that we address in this dissertation: how to specify and verify programs that use inclusion polymorphism.

Conventional specification techniques are poorly matched to programs and functions that use inclusion polymorphism, because they do not exploit the implicit association of instance operations with objects.

We want to specify functions that use inclusion polymorphism in a way that parallels code that uses inclusion polymorphism. For example, the specification in Figure 1.4 corresponds closely to the Trellis/Owl code for `sumFirst` given above, since it is written as if all arguments must have type `IntPair`. But we also allow instances of subtypes of `IntPair`, such as `IntTriple`, to be passed as arguments.

The effect of `sumFirst` is described in Figure 1.4 using terms that are tailored to the type `IntPair`. That is, we think of the *abstract value* of an `IntPair` as a pair of integers. A set of abstract values is described by functions, such as the pairing function "$\langle , \rangle$" and the postfixed function ".first," which returns the first component of a pair.

However, when the arguments have type `IntTriple`, there is a problem understanding the description of the effect of `sumFirst`. To illustrate this problem, we describe the abstract values of `IntTriple` as sequences of three integers, such as $\langle 4, 5, 6 \rangle$, with postfixed selector functions "[1]," "[2]," and "[3]." Consider the following call to `sumFirst`:

```
sumFirst(make(IntPair,1,2), make(IntTriple,4,5,6)).
```

The abstract value of `make(IntTriple,4,5,6)` is the sequence $\langle 4, 5, 6 \rangle$. However, since the specification of `sumFirst` uses the type `IntPair`, we cannot describe the result by substituting the abstract value $\langle 4, 5, 6 \rangle$ into the description of the effect of `sumFirst`, as the term "$\langle 4, 5, 6 \rangle$.first" is meaningless (since we can only apply ".first" to the abstract values of `IntPair`).

Figure 1.4: Specification of the function `sumFirst`.

```
fun sumFirst(p1,p2: IntPair) returns(i:Int)
    effect i = p1.first + p2.first
```

To give a formal semantics to specifications that allows arguments of all subtypes of the specified formal argument types, we must also have a formal definition of when one abstract type is a subtype of another.

Of the formal definitions of "subtype" that have appeared in the literature[1], only the work of Bruce and Wegner [BW87] has a definition of subtyping among abstract data types. However, Bruce and Wegner's definitions do not handle incompletely specified and nondeterministic types, both of which are important in practical use. (We discuss Bruce and Wegner's work and the work of others in more detail in Chapter 5.)

Conventional techniques for program verification also need to be adapted for verifying programs that use inclusion polymorphism. One problem is illustrated by our difficulties reasoning about the call to `sumFirst` above; that is, we need a way to coerce the abstract value of a subtype into an abstract value of its supertypes. Another verification problem arises because often we only know that an expression of a given type `T` denotes an instance of some subtype of `T`. We must ensure that reasoning about such expressions as if they denoted instances of type `T` does not lead to invalid conclusions.

1.4 Overview of Our Solution

Our ideas for solving the specification and verification problems described in the previous section are based on concepts used by programmers working in languages with generic invocation mechanisms. We simplify the problem by only dealing with an applicative programming language and with immutable types. (An *immutable type* is an abstract type whose instances have no time-varying state.)

The fundamental concept is the notion of a subtype. Informally, each instance of a subtype behaves like some instance of its supertypes. To formalize this notion, we formalize the implicit notions of "type" and "behavior."

Abstract types are described by specifications. The implementations of type specifications are formally modeled with a family of algebras.

The behavior of an instance is determined by the way it affects the output of code run in an environment that binds the instance to some identifier used in that code. Since instances can interact, we speak of the behavior of the environment itself, and how it

[1]A survey can be found in [DT88].

affects the output of code that uses the identifiers defined by the environment. The formal model of such code is called an observation. For example, `sumFirst(p1,p2)` defines an observation on environments where `p1` : `IntPair` and `p2` : `IntPair` are defined.

We say that one environment imitates another when the first environment behaves like the second, with respect to each observation of interest. For example, an environment η_1 such that η_1(`p2`) is an `IntTriple` with abstract value $\langle 4,5,6\rangle$ imitates an environment η_2, where η_2(`p2`) is an `IntPair` with abstract value $\langle 4,5\rangle$ with respect to the set of observations defined by `first(p2)` and `second(p2)`.

We use imitation to evaluate the assertions in specifications as follows. We assume that we can evaluate assertions in an environment where for each type T, each identifier of type T denotes an instance of type T. We call such an environment a *nominal environment.* To evaluate an assertion in an environment that is not nominal, one finds a nominal environment that the first environment imitates and then evaluates the assertion in the nominal environment. For example, the assertion "p2.first = 4" holds in the environment η_1 described above, because η_1 imitates the nominal environment η_2 described above, and in η_2 "p2.first" is 4.

We define subtyping so that one can always evaluate the assertions in our specifications. Since our specifications only allow instances of subtypes as arguments, the environments created by binding the actual arguments to the formals are such that each identifier of some type `T` denotes an object of some subtype of `T`. Our definition of subtyping ensures that each such environment imitates some nominal environment. Since one can always find a nominal environment, one can always evaluate the assertions in our specifications. For example, one can pass arguments of type `IntTriple` to `sumFirst`, because one can always find instances of `IntPair` that the instances of `IntTriple` imitate.

We can prove subtype relationships by showing that every instance of each subtype simulates some instance of its supertypes. Simulation is preserved by each generic invocation, whereas imitation is only guaranteed to preserve externally observable behavior. For example, an instance of `IntTriple` whose abstract value is $\langle 4,5,6\rangle$ simulates an instance of `IntPair` whose abstract value is $\langle 4,5\rangle$, since the result of applying the operations `first` and `second` to the `IntTriple` simulate the results of applying these operations to the `IntPair`. (For the built-in types like `Int`, simulation means equality.)

We use simulation relationships during program verification to translate the abstract values of a subtype into the abstract values of a supertype. For example, to find the effect of

```
sumFirst(make(IntPair,1,2), make(IntTriple,4,5,6)),
```

one proceeds as follows. The abstract value of `make(IntTriple,4,5,6)` is the sequence $\langle 4,5,6\rangle$. This instance simulates an instance of `IntPair` whose abstract value is $\langle 4,5\rangle$, so one takes $\langle 4,5\rangle$ as the abstract value of the actual. One can then substitute the abstract values of the actuals into the description of the effect, obtaining

$$\mathrm{i} = \langle 1,2\rangle.\mathrm{first} + \langle 4,5\rangle.\mathrm{first}$$

which can be simplified to "i = 5."

In the rest of this dissertation we not only work out the above solution in detail, but we also show how to treat more interesting examples. In particular, we show how to handle incompletely specified and nondeterministic types. Incomplete specifications are important, because they allow a designer to leave implementation decisions open. Nondeterminism is important for some applications and also can be used to leave implementation decisions open.

1.5 Plan of the Dissertation

In this section we describe where the above concepts are discussed in this dissertation.

Chapters 2 and 3 provide context and background. In Chapter 2 we describe algebraic models of specifications and specifications for abstract types. In Chapter 3 we describe observations and how they are described by a programming language with a generic invocation mechanism.

In Chapter 4 we describe a technique for the specification of functions and programs that use inclusion polymorphism. A formal semantics of such specifications is given that uses the "imitates" relation between environments.

In Chapter 5 we give a formal definition of subtype relationships. We show that this definition ensures that the specifications described in Chapter 5 are sensible.

In Chapter 6 we give a formal definition of simulation. We show how to use simulation relationships to prove subtype relationships.

In Chapter 7 we show how simulation can be used in program verification. Simulation relationships are used to translate assertions from subtypes to supertypes. These translations are used in a Hoare-style proof system for the applicative language introduced in Chapter 3.

In Chapter 8 we discuss how our results can be applied in other programming languages; we also discuss extensions and future work. In Chapter 9 we offer a summary and some conclusions. In particular, we discuss lessons for programmers and language designers.

In Appendix A we summarize the notation and definitions introduced throughout the dissertation. In Appendix B we describe the built-in types of the type specifications described in Chapter 2. In Appendix C we give the semantics of the recursively-defined functions of the programming language described in Chapter 3; we also show that these functions preserve simulation relationships.

Chapter 2

Algebraic Models of Type Specifications

The primary purpose of this chapter is to define the semantics of immutable type specifications, for use in our definition of subtype relations. We take a "loose" view of type specifications; that is, the semantics of a type specification is a set of algebraic models. We also describe a specification language that we use in examples and for program verification.

In what follows we first define algebras. We then describe our specification language and its semantics. Afterwards we discuss how nondeterminism and exceptions are handled.

2.1 Algebras

We use algebras to model specifications of immutable abstract types. Our algebras are an extension of the relational structures studied by Nipkow [Nip86] [Nip87]. These structures consist of a carrier set for each type and a set of nondeterministic operations. To Nipkow's relational structures we have added functions, called trait functions, that aid in the evaluation of assertions [Win83, Chapter 2]. As we explain algebras, we point out some differences from Nipkow's notation.

The operations of an algebra are abstractions of the procedures of the classes that implement abstract types in object-oriented programs. To model nondeterministic procedures, the operations of an algebra are set-valued functions. That is, an operation of an algebra returns the set of the possible results of the corresponding procedure. To model procedure calls that do not halt or that encounter run-time errors, we use the special value $\bot$. So a procedure that might either return 1 or never halt on some argument q would be modeled by an operation that, when called with q, has $\{1, \bot\}$ as its set of possible results.

The operations of an algebra are named by *operation symbols* of the form $\mathbf{g}_{\vec{\mathtt{S}}\to\mathtt{T}}$. The signatures used as subscripts on the operation symbols list the types of the arguments and the type of the result for the algebra's operation. Unlike the procedures of a class, which may be polymorphic if they use generic invocation, the operations of an algebra are not polymorphic. We model a generic invocation by subscripting the *generic operation symbol* used in a program (e.g., **g**) with the types of the actual arguments, and consulting the appropriate operation of the algebra to find the set of possible results (see Chapter 3 for details). While it might be more elegant to have algebras model generic invocation, we chose to model generic invocation in the semantics of our programming language, so that our algebraic models of type specifications more closely resemble standard algebras.

In addition to its operations, an algebra also has a set of trait functions. These functions have no counterpart in the classes that implement abstract types. The trait functions are specified by traits, which are descriptions of sets of abstract values [GH86b], called *sorts*.

In our formal models, we do not have separate representations for abstract values and for objects. That is, we identify the objects of a type with their abstract values. So each type symbol is also the name of a sort. Since some sorts are described using other sorts, in general there may be more sorts than there are types.

There is a small set of types in an algebra called the *visible types*. These types, such as `Bool` and `Int`, are used to define observations.

For each sort T in some set *SORTS* of sort symbols, an algebra A has a set, $A_{\mathtt{T}}$, called the *carrier set of* T. If $q \in A_{\mathtt{T}}$, then we say that *q has sort* T; furthermore, if T is also type, we say that *q has type* T or that *q is an instance of type* T. An element of a carrier set, often written o, q, or r, is also called an *object*. Each sort's carrier set contains the element $\bot$. An element of a carrier set is called *proper* if it is not $\bot$. So that each proper object has only one sort, the carrier sets of an algebra must be disjoint, except that $\bot$ is in each of them. So the *carrier set of an algebra A*, written $|A|$, is a family of sets indexed by sort symbols, that are disjoint except that each contains $\bot$. The family of carrier sets indexed by type symbols is written A_{TYPES}.

An *I-indexed set* is a surjective function from an index set I to the elements of a set.

For example, A_{TYPES} is a $TYPES$-indexed family of sets, since

$$A_{TYPES} \overset{\mathrm{def}}{=} \{A_{\mathrm{T}} \mid \mathrm{T} \in TYPES\}. \tag{2.1}$$

An *operation* on a carrier set is a mapping that takes a tuple of zero or more elements of a carrier set and returns a nonempty set of possible results. A *possible result* is simply an element of a carrier set. We also regard operations as binary relations, where the set of possible results of an application of a binary relation f to a tuple of arguments $\vec{q}$ is

$$f(\vec{q}) \overset{\mathrm{def}}{=} \{r \mid (\vec{q}, r) \in f\}. \tag{2.2}$$

As above, we often use vector notation for tuples. For example, the notation $\vec{q}$ stands for a n-tuple $\langle q_1, \ldots, q_n \rangle$, where $n \geq 0$. We say that $\vec{q}$ has type $\vec{\mathrm{S}}$, written $\vec{q} \in A_{\vec{\mathrm{S}}}$, if each q_i has type S_i. The notation $A_{\vec{\mathrm{S}}}$ stands for $A_{\mathrm{S}_1} \times \cdots \times A_{\mathrm{S}_n}$, where if $\vec{\mathrm{S}} = \langle\rangle$, then $A_{\vec{\mathrm{S}}} \overset{\mathrm{def}}{=} \{\langle\rangle\}$. The same notation is also used for sorts.

Each operation in an algebra has a name and a signature. If $TYPES$ is a set of type symbols, then an element of $(TYPES^* \times TYPES)$ is called a *signature* and is written as $\rightarrow \mathrm{T}$ or $\mathrm{S}_1, \ldots, \mathrm{S}_n \rightarrow \mathrm{T}$ or $\vec{\mathrm{S}} \rightarrow \mathrm{T}$. An *operation symbol* has its signature as a subscript, for example, $\mathbf{g}_{\vec{\mathrm{S}} \rightarrow \mathrm{T}}$. A *family of typed operation symbols* is a $(TYPES^* \times TYPES)$-indexed family of sets. If OPS is a family of typed operation symbols, then we abbreviate the statement that $\mathbf{g}_{\vec{\mathrm{S}} \rightarrow \mathrm{T}}$ is an element of $OPS_{\vec{\mathrm{S}} \rightarrow \mathrm{T}}$ by writing $\mathbf{g}_{\vec{\mathrm{S}} \rightarrow \mathrm{T}} : \vec{\mathrm{S}} \rightarrow \mathrm{T}$.

A *set of typed operations* on A_{TYPES} is a set of operations indexed by a family of typed operation symbols, with the following property: for each operation $A_{\mathbf{g}_{\vec{\mathrm{S}} \rightarrow \mathrm{T}}}$ and for each tuple $\vec{q} \in A_{\vec{\mathrm{S}}}$, $A_{\mathbf{g}_{\vec{\mathrm{S}} \rightarrow \mathrm{T}}}(\vec{q})$ is a nonempty subset of A_{T}.

A function (or operation) is *strict* if whenever one of its arguments is $\bot$, then the (only possible) result is $\bot$. A function (or operation) is *total* if whenever all its arguments are proper (i.e., not $\bot$), then no (possible) result is $\bot$. A function (or operation) that is not total is *partial.*

A *trait function* is a strict and total mapping that takes a tuple of zero or more elements of a carrier set and returns an element of the carrier set.

A *family of sorted trait function symbols* is a $(SORTS^* \times SORTS)$-indexed set.

A *set of sorted trait functions* on $|A|$, is a set of trait functions indexed by a family of sorted trait function symbols with the following property: for each trait function $A_{\mathrm{f}} : \vec{\mathrm{S}} \rightarrow \mathrm{T}$ and for each tuple $\vec{q} \in A_{\vec{\mathrm{S}}}$, $A_{\mathrm{f}}(\vec{q}) \in A_{\mathrm{T}}$.

Definition 2.1.1 (algebra). An *algebra* $A = (|A|, A_{TYPES}, A_{TFUNS}, A_{OPS})$, consists of:

- a carrier set, $|A|$,
- a family of carrier sets for each type, A_{TYPES},
- a set, A_{TFUNS}, of sorted trait functions on $|A|$, and
- a set, A_{OPS}, of typed operations on A_{TYPES}.

Besides the addition of trait functions, our algebras differ from Nipkow's because Nipkow does not use $\perp$ to represent nontermination. Instead Nipkow describes each operation by a binary relation giving its input/output behavior for proper elements and a termination set that describes the inputs for which termination is guaranteed. As Nipkow suggests, we can translate from his notation into ours [Nip87, Page 9]. However, the operations of our algebras can be non-strict, while the operations of Nipkow's algebras must be strict. Non-strict operations are useful for modeling types with lazy evaluation, such as streams.

We classify operations and algebras as follows [Nip87, Page 9]. If an application of an operation has a single possible result, it is *deterministic*. An operation is deterministic if all applications of that operation are deterministic. An operation that is not deterministic is *nondeterministic*. An algebra is total or deterministic if all its operations have that property. An algebra is partial or nondeterministic if some of its operations are partial or nondeterministic.

An example algebra, B, is presented in Figure 2.1 (on page 26). The figure first defines the carrier sets for each type, then the trait functions, and finally the possible results of the operations. The carrier set for the type `Bool` contains $\perp$, *true* and *false*. The only proper element of the carrier set for `BoolClass` is *Bool*, which is used like a class object in an object-oriented language. The trait functions symbols contain sharp signs (#) as place holders for arguments, allowing prefix and infix syntax for invocations. The operations themselves include the nullary operation named $\texttt{Bool}_{\rightarrow\texttt{BoolClass}}$ which has the "class object" *Bool* as its only possible result, class operations for true and false, and instance operations for the usual logical operations. We abbreviate the description of trait functions and operations by the following conventions.

- A variable such as b only stands for a proper element of the appropriate carrier set, never for $\perp$.
- The only omitted cases involve $\perp$ as an argument, and for these the only possible result is $\perp$.
- The trait functions are used to define the operations.

Notice that all the operations of B in Figure 2.1 are strict, total, and deterministic. Hence B is a total and deterministic algebra.

Each algebra has a signature, which is defined below. Following Nipkow, we include in the signature a set of *visible types*, which are used to define observations.

Definition 2.1.2 (signature). If $A = (|A|, A_{TYPES}, A_{TFUNS}, A_{OPS})$ is an algebra, then its *signature*, $SIG(A) = (SORTS, TYPES, V, TFUNS, OPS)$, consists of:

- the set, $SORTS$, of sort symbols that index the carrier set of A,
- a nonempty set, $TYPES \subseteq SORTS$, of the type symbols that index A_{TYPES},
- a nonempty set, $V \subseteq TYPES$ of *visible types*,
- the set, $TFUNS$, of sorted trait function symbols that index the trait functions of A, and
- the set, OPS, of typed operation symbols that index the operations of A.

In other chapters we often abbreviate signatures by omitting the set of sort symbols $SORTS$.

If $SIG(A) = \Sigma$, then A is a Σ-*algebra.* We call

$$\Sigma' = (SORTS', TYPES', V', TFUNS', OPS')$$

a *subsignature* of

$$\Sigma = (SORTS, TYPES, V, TFUNS, OPS)$$

if $SORTS' \subseteq SORTS$, $TYPES' \subseteq TYPES$, $V' \subseteq V$, $TFUNS' \subseteq TFUNS$, and $OPS' \subseteq OPS$ [EM85, Section 6.8]. By $TFUNS' \subseteq TFUNS$, we mean that the subset relationship holds at each index.

Figure 2.1: An algebra B for the Booleans.

Carrier Sets

$$\begin{aligned}
B_{\texttt{Bool}} &\stackrel{\text{def}}{=} \{\bot, \mathit{true}, \mathit{false}\} \\
B_{\texttt{BoolClass}} &\stackrel{\text{def}}{=} \{\bot, \mathit{Bool}\}
\end{aligned}$$

Trait Functions

$$\begin{aligned}
B_{\text{Bool}}() &\stackrel{\text{def}}{=} \mathit{Bool} \\
B_{\text{true}}() &\stackrel{\text{def}}{=} \mathit{true} \\
B_{\text{false}}() &\stackrel{\text{def}}{=} \mathit{false} \\
B_{\neg\#}(b) &\stackrel{\text{def}}{=} \begin{cases} \mathit{false} & \text{if } b = \mathit{true} \\ \mathit{true} & \text{if } b = \mathit{false} \end{cases} \\
B_{\#\&\#}(b_1, b_2) &\stackrel{\text{def}}{=} \begin{cases} \mathit{true} & \text{if } b_1 = b_2 = \mathit{true} \\ \mathit{false} & \text{otherwise} \end{cases} \\
B_{\#|\#}(b_1, b_2) &\stackrel{\text{def}}{=} \begin{cases} \mathit{false} & \text{if } b_1 = b_2 = \mathit{false} \\ \mathit{true} & \text{otherwise} \end{cases} \\
B_{\#\Rightarrow\#}(b_1, b_2) &\stackrel{\text{def}}{=} \begin{cases} \mathit{false} & \text{if } b_1 = \mathit{true} \text{ and } b_2 = \mathit{false} \\ \mathit{true} & \text{otherwise} \end{cases} \\
B_{\#\equiv\#}(b_1, b_2) &\stackrel{\text{def}}{=} \begin{cases} \mathit{true} & \text{if } b_1 = b_2 \\ \mathit{false} & \text{otherwise} \end{cases}
\end{aligned}$$

Operations

$$\begin{aligned}
B_{\texttt{Bool}_{\to\texttt{BoolClass}}}() &\stackrel{\text{def}}{=} \{\mathit{Bool}\} \\
B_{\texttt{true}_{\texttt{BoolClass}\to\texttt{Bool}}}(\mathit{Bool}) &\stackrel{\text{def}}{=} \{\mathit{true}\} \\
B_{\texttt{false}_{\texttt{BoolClass}\to\texttt{Bool}}}(\mathit{Bool}) &\stackrel{\text{def}}{=} \{\mathit{false}\} \\
B_{\texttt{not}_{\texttt{Bool}\to\texttt{Bool}}}(b) &\stackrel{\text{def}}{=} \{B_{\neg\#}(b)\} \\
B_{\texttt{and}_{\texttt{Bool},\texttt{Bool}\to\texttt{Bool}}}(b_1, b_2) &\stackrel{\text{def}}{=} \{B_{\#\&\#}(b_1, b_2)\} \\
B_{\texttt{or}_{\texttt{Bool},\texttt{Bool}\to\texttt{Bool}}}(b_1, b_2) &\stackrel{\text{def}}{=} \{B_{\#|\#}(b_1, b_2)\}
\end{aligned}$$

Let $\Sigma V = (V, V, V, \mathit{TFUNS}_V, \mathit{OPS}_V)$ be a fixed signature for the visible types. If ΣV is a subsignature of $\mathit{SIG}(A) = (\mathit{SORTS}, \mathit{TYPES}, V', \mathit{TFUNS}, \mathit{OPS})$, then the *$\Sigma V$-reduct of A* is the algebra

$$A_{(\Sigma V)} \stackrel{\text{def}}{=} \left(\begin{array}{l} \{A_{\mathrm{T}} \mid \mathrm{T} \in V\}, \{A_{\mathrm{T}} \mid \mathrm{T} \in V\}, \\ \{A_{\mathrm{f}} \mid \mathrm{f} \in \mathit{TFUNS}_V\}, \{A_{\mathrm{g}_{\vec{\mathrm{S}} \to \mathrm{T}}} \mid \mathrm{g}_{\vec{\mathrm{S}} \to \mathrm{T}} \in \mathit{OPS}_V\} \end{array} \right) \tag{2.3}$$

[EM85, Section 6.8]. That is, $A_{(\Sigma V)}$ has as its carrier sets the carrier sets of the visible types in A and as its trait functions and operations those named in ΣV.

We assume that the visible types and their semantics are fixed by convention, that is, by one's programming or specification language. This assumption allows us to compare the results of observations, since the possible results of an observation must be instances of visible types. We state this assumption formally as follows. Let B be a fixed algebra whose signature is ΣV. Formally, our assumption is that for each Σ-algebra A, the fixed signature ΣV is a subsignature of Σ, the set of visible types in Σ is V (the same as in ΣV), and the ΣV-reduct of A is the fixed algebra B. In the next section we describe this convention for our specification language.

The ***semantics of a specification*** is a set of algebras with the same signature. Each algebra in the semantics of a specification named *SPEC* is called a *SPEC-algebra*. The following sections present one way of describing such sets.

2.2 Type Specifications

In this section we describe our specification language, which is adapted from Wing's interface specification language for CLU [Win83] [LG86, Chapter 10] [GHW85] [Win87]. We have modified Wing's syntax, in part to distinguish class operations from instance operations. Unlike Wing, we deal only with immutable types.

For us, a specification has the following goals:

- describing a set of algebras, and
- describing an interface for programmers that consists of a set of generic operation symbols and their nominal signatures.

The algebras we specify have many characteristics of Trellis/Owl classes, such as operations that return class objects and operations that model class and instance operations.

A generic operation symbol may be present in many different type specifications. We use this convention to support generic invocation. For example, we specify **`first`** as an operation of both `IntPair` and `IntTriple`.

Each of occurrence of a generic operation symbol in a specification is associated with a different *nominal signature*, which is a pair consisting of a tuple of type symbols and a type symbol, written $\rightarrow \mathtt{T}$ or $\mathtt{S}_1, \ldots, \mathtt{S}_n \rightarrow \mathtt{T}$ or $\vec{\mathtt{S}} \rightarrow \mathtt{T}$. The nominal signatures of generic operation symbols are used in type-checking and reasoning about programs.

Abstractly, a specification has four parts: a set of type symbols, a binary relation on these type symbols, a set of traits, and a specification of the operations of a model. We call the binary relation on types a *presumed subtype relation*, since it should relate subtypes to supertypes. Each trait describes the abstract values of a type; formally it specifies the carrier set and trait functions of a model. (Unless these traits are unconventional, they will be found in the Larch Shared Language Handbook [GH86a].) The bulk of our specifications is taken up by operation specifications.

An example specification, named IPT, is found in Figure 2.2. We will use the specification IPT to help explain our specification language in this section. The trait "ThreeSeq" that describes the abstract values of `IntTriple` is found in Figure 2.3.

We first describe the syntax of specifications, then how a specification determines the syntactic interfaces of the algebras in its semantics, and finally how a specification determines the behavior of these algebras.

2.2.1 Type Specification Syntax

The syntax of type specifications is given in Figure 2.4. The nonterminals ⟨type⟩, ⟨trait function⟩, and ⟨identifier⟩ represent type symbols, trait function symbols, and generic operation symbols and other identifiers (respectively).

A specification consists of a list of ⟨type spec⟩s, each of which specifies an abstract type. A ⟨type spec⟩ has a type name, lists of the generic operation symbols that name the type's class and instance operations, a ⟨basis⟩ clause, and a list of operation specifications. The ⟨basis⟩ clause tells what trait is used in the operation specifications. Operations can be specified to be deterministic or nondeterministic by using either **op** or **ndop**. Each ⟨op spec⟩ also lists the operation's nominal signature.

Figure 2.2: The type specification IPT, containing **IntPair** and **IntTriple**.

```
IntPair immutable type
    class ops [make] instance ops [first, second]
    based on sort C from Pair with [Int for T1, T2]

    op make(c:IntPairClass, f,s:Int) returns(p:IntPair)
        effect p = ⟨f,s⟩

    op first(p:IntPair) returns(i:Int)
        effect i = p.first

    op second(p:IntPair) returns(i:Int)
        effect i = p.second


IntTriple immutable type IntPair
    class ops [make] instance ops [first, second, third]
    based on sort C from ThreeSeq with [Int for T]

    op make(c:IntTripleClass, f,s,t:Int) returns(r:IntTriple)
        effect r = ⟨f,s,t⟩

    op first(t:IntTriple) returns(i:Int)
        effect i = t[1]

    op second(t:IntTriple) returns(i:Int)
        effect i = t[2]

    op third(t:IntTriple) returns(i:Int)
        effect i = t[3]
```

Figure 2.3: The trait ThreeSeq that describes the abstract values of `IntTriple`.

ThreeSeq: **trait**
  introduces
    $\langle\ \#, \#, \#\ \rangle$: T,T,T $\rightarrow$ C
    #[1]: C $\rightarrow$ T
    #[2]: C $\rightarrow$ T
    #[3]: C $\rightarrow$ T
  asserts C **generated by** $[\langle \#, \#, \# \rangle]$
    C **partitioned by** [[1], [2], [3]]
    for all [f,s,t: T]
      $\langle f, s, t\rangle[1] = f$
      $\langle f, s, t\rangle[2] = s$
      $\langle f, s, t\rangle[3] = t$
    implies converts [[1], [2], [3]]

The behavior of an operation is described using a ⟨requires clause⟩ and an ⟨effects clause⟩. These clauses contain boolean ⟨term⟩s written using trait function symbols, the formal arguments of the operation, and equations. These terms must sort-check in the sense that trait functions can only be applied to terms of the expected sorts and equations are only allowed between terms of the same sort.

In an ⟨op spec⟩ for an instance operation that appears in the specification of a type named `T`, the first argument's type must be specified to be `T`. Similarly, the first argument of a class operation of a type `T` must be specified to be `TClass`, where `TClass` is formed by joining the suffix `Class` to the type name `T`.

For convenience, we define the following syntactic sugars. An ⟨trait function⟩ such as "f" used in a ⟨term⟩ without arguments is syntactic sugar for "f()." A declaration such as "f,s: Int" is syntactic sugar for the declaration list "f: Int, s: Int." Furthermore, an omitted ⟨requires clause⟩ is syntactic sugar for a requires clause of the form "**requires** true." A syntactic sugar for exceptions is discussed in the last section of this chapter.

2.2.2 Syntactic Interfaces of Type Specifications

A specification describes two syntactic interfaces. The first interface is a set of generic operation symbols and their *nominal* signatures. The second interface is the signature of the algebras that satisfy the specification. The signature of the algebras that satisfy

Figure 2.4: Syntax of Specifications.

⟨specification⟩ ::= ⟨type spec list⟩
⟨type spec list⟩ ::= ⟨type spec⟩ | ⟨type spec list⟩ ⟨type spec⟩

⟨type spec⟩ ::= ⟨type⟩ **immutable type**
    ⟨class ops⟩ ⟨instance ops⟩
    ⟨basis⟩
    ⟨op spec list⟩

⟨class ops⟩ ::= ⟨empty⟩ | **class ops** [⟨ident list⟩]
⟨instance ops⟩ ::= **instance ops** [⟨ident list⟩]
⟨empty⟩ ::=
⟨ident list⟩ ::= ⟨identifier⟩ | ⟨ident list⟩ , ⟨identifier⟩

⟨basis⟩ ::= **based on sort** ⟨identifier⟩ **from** ⟨identifier⟩ ⟨with clause⟩
⟨with clause⟩ ::= ⟨empty⟩ | **with** [⟨renaming list⟩]
⟨renaming list⟩ ::= ⟨renaming⟩ | ⟨renaming list⟩ , ⟨renaming⟩
⟨renaming⟩ ::= ⟨identifier⟩ **for** ⟨ident list⟩

⟨op spec list⟩ ::= ⟨op spec⟩ | ⟨op spec list⟩ , ⟨op spec⟩
⟨op spec⟩ ::= ⟨op kind⟩ ⟨op signature⟩
    ⟨requires clause⟩ ⟨effect clause⟩
⟨op kind⟩ ::= **op** | **ndop**
⟨op signature⟩ ::= ⟨identifier⟩ (⟨decl list⟩) **returns** (⟨decl⟩)
⟨decl list⟩ ::= ⟨decl⟩ | ⟨decl list⟩ , ⟨decl⟩
⟨decl⟩ ::= ⟨identifier⟩ : ⟨type⟩

⟨requires clause⟩ ::= **requires** ⟨term⟩
⟨effect clause⟩ ::= **effect** ⟨term⟩

⟨term⟩ ::= ⟨identifier⟩
    | ⟨trait function⟩ (⟨term list⟩) | ⟨trait function⟩ ( )
    | ⟨term⟩ = ⟨term⟩ | (⟨term⟩)
⟨term list⟩ ::= ⟨term⟩ | ⟨term list⟩ , ⟨term⟩

a specification is partly determined by the set of generic operation symbols and their nominal signatures.

The set of generic operation symbols of a specification consists of the symbols following **op** or **ndop** in ⟨op spec⟩s, all type symbols that are not class types, and generic operation symbols for the visible types. For example, the set of generic operation symbols of IPT includes `make`, `first`, `second`, `third`, `IntPair`, `IntTriple`, `Bool`, `Int`, `IntStream`, `BoolStream`, and generic operation symbols for the visible types such as `true`, `false`, `not`, `and`, `or`, `add`, and so on.

The set of type symbols described by a specification consists of the type symbols named at the beginning of each ⟨type spec⟩, the *visible types* `Bool`, `Int`, `IntStream`, and `BoolStream`, whatever `OneOf` types are necessary (as described in the section on exceptions below), and a *class type* for each of the types already mentioned, formed by adding "`Class`" as a suffix to each of the other type symbols. For example, the set of type symbols of the specification IPT includes `IntPair`, `IntTriple`, `IntPairClass`, and `IntTripleClass`, in addition to the visible types and their associated class types: `BoolClass`, `IntClass`, `IntStreamClass`, and `BoolStreamClass`. A type symbol such as `IntPair` is a generic operation symbol with no arguments.

The generic operation symbols associated with the visible types are found by taking the operations associated with those types (see Figures 2.1, B.1, and B.2) and stripping the associated operation symbols of their subscripted signatures. For example, in Figure 2.1, the operation $\mathtt{not}_{\mathtt{Bool}\to\mathtt{Bool}}$ is defined, which means that a generic operation symbol `not` *is associated with* `Bool`.

We summarize the association of generic operation symbols and nominal signatures in a specification with a set-valued function, called a *nominal signature map*. It takes a generic operation symbol and returns a nonempty set of nominal signatures. If *NomSig* is a nominal signature map and $(\vec{\mathtt{S}} \to \mathtt{T}) \in NomSig(\mathtt{g})$, then g *has nominal signature* $\vec{\mathtt{S}} \to \mathtt{T}$.

A specification determines a nominal signature map, *NomSig*, as follows. For each generic operation symbol of the specification g, a nominal signature is an element of *NomSig*(g) if and only if either:

- the nominal signature is $\vec{\mathtt{S}} \to \mathtt{T}$ and g is associated with a visible type because $\mathtt{g}_{\vec{\mathtt{S}}\to\mathtt{T}}$

is a typed operation symbol from Figure 2.1, B.1, B.2, or the analogous algebra for `BoolStream` (see Appendix B),

- the nominal signature is $\rightarrow$ **`gClass`**, the specification includes **g** as a type symbol; and **`gClass`** is the type symbol formed by attaching "**`Class`**" as a suffix of **g**, or
- the nominal signature is $\vec{\mathtt{S}} \rightarrow \mathtt{T}$, and **g** is a class or instance operation specified with operation signature:

$$(\vec{\mathtt{x}} : \vec{\mathtt{S}})\ \mathbf{returns}(\mathtt{y} : \mathtt{T}).$$

For example, if *NomSig* is the nominal signature map of IPT, then we have:

$$\begin{aligned}
\mathit{NomSig}(\mathtt{not}) &= \{\mathtt{Bool} \rightarrow \mathtt{Bool}\} \\
\mathit{NomSig}(\mathtt{IntPair}) &= \{\rightarrow \mathtt{IntPairClass}\} \\
\mathit{NomSig}(\mathtt{make}) &= \{\mathtt{IntPairClass}, \mathtt{Int}, \mathtt{Int} \rightarrow \mathtt{IntPair}, \\
&\quad \mathtt{IntTripleClass}, \mathtt{Int}, \mathtt{Int}, \mathtt{Int} \rightarrow \mathtt{IntPair}, \} \\
\mathit{NomSig}(\mathtt{first}) &= \{\mathtt{IntPair} \rightarrow \mathtt{Int}, \\
&\quad \mathtt{IntTriple} \rightarrow \mathtt{Int}\}.
\end{aligned}$$

We require that for each generic operation symbol **g**, the nominal signatures in *NomSig*(**g**) with the same number of argument positions all differ in their first argument position, and there is at most one element of *NomSig*(**g**) with no argument types.

The signature, $\Sigma = (\mathit{SORTS}, \mathit{TYPES}, V, \mathit{TFUNS}, \mathit{OPS})$, determined by a specification *SPEC*, has as its set of type symbols, *TYPES*, the set of type symbols of *SPEC*, and as its set of visible types

$$V = \{\mathtt{Bool}, \mathtt{Int}, \mathtt{BoolStream}, \mathtt{IntStream}\}. \tag{2.4}$$

The set of sorts *SORTS* and the set of sorted trait function symbols, *TFUNS*, are determined by the traits referenced in the ⟨basis⟩ clauses of *SPEC*. These traits introduce various sorts and operations, which become the sorts and trait function symbols, as renamed by the ⟨renaming⟩ clause. In addition to these, we add a trait for each class type **`TClass`** of the form

TClass: **trait**

introduces T: $\rightarrow$ TClass.

The sort name following the keyword **sort** in the specification of a type named `T` is renamed to `T`. The signature of each trait function symbol from a trait referenced in the ⟨basis⟩ clause of the specification of a type `T` has the sort name following the keyword **sort** replaced by `T`, in addition to the other renamings. The set of operations *OPS* is determined by attaching to each generic operation symbol **g** each subscript from *NomSig*(**g**). That is,

$$OPS \stackrel{\text{def}}{=} \left\{ \mathrm{g}_{\vec{\mathrm{S}} \rightarrow \mathrm{T}} \mid (\mathrm{g} \in GOP) \wedge ((\vec{\mathrm{S}} \rightarrow \mathrm{T}) \in NomSig(\mathrm{g})) \right\}. \tag{2.5}$$

For example, the signature of an IPT-algebra has operation symbols $\texttt{first}_{\texttt{IntPair}\rightarrow\texttt{Int}}$ and $\texttt{first}_{\texttt{IntTriple}\rightarrow\texttt{Int}}$, because `first` is a generic operation symbol and *NomSig*(`first`) consists of `IntPair` $\rightarrow$ `Int` and `IntTriple` $\rightarrow$ `Int`.

If *SPEC* is a specification, then we write *SIG*(*SPEC*) for its signature.

2.2.3 Satisfaction for Type Specifications

Besides having the right signature, to *satisfy a specification* an algebra must satisfy both the specification's traits and its operation specification, and its model of the visible types must be standard.

Since the traits used in a specification do not specify the operations of a model, we can use a standard definition of satisfaction for traits, by extracting the carrier set and trait functions from an algebra.

Let $A = (|A|, A_{TYPES}, A_{TFUNS}, A_{OPS})$ be an algebra. Then *the functional structure of A* is formed by deleting $\perp$ from each sort's carrier set, restricting the trait functions to these domains, and throwing out A_{TYPES} and A_{OPS}. The trait functions are defined on carrier sets without $\perp$ because they are strict and total. A functional structure is thus a multi-sorted algebra, as used in the semantics of first-order logic.

We say that *A satisfies* the traits of a specification if and only if the functional structure of *A* is a model of those traits.

We evaluate the terms in a requires clause or an effects clause in the standard way, using extended assignments. An *assignment* is an map from a set of sorted identifiers to the carrier set of an algebra that maps each identifier of sort `T` to a proper element of sort T. We can extend such an assignment to an evaluation of terms whose free identifiers are in the domain of the assignment by using the trait functions of the algebra in the assignment's range to evaluate trait function symbols and by using the assignment itself to evaluate free identifiers [EM85, Section 1.10]. If $\eta : X \rightarrow |A|$ is such an assignment, we write $\overline{\eta}$ for its extension to a mapping from terms to elements of the carrier set of A. For example, if $\eta(\mathrm{p}) = \langle 1, 2\rangle$, and $A_{\#.\mathrm{first}}(\langle i, j\rangle) = i$, then

$$\overline{\eta}[\![\mathrm{p.first}]\!] = A_{\#.\mathrm{first}}(\eta(\mathrm{p})) = A_{\#.\mathrm{first}}(\langle 1,2\rangle) = 1.$$

We further extend assignments so that

$$\overline{\eta}[\![E_1 = E_2]\!] \overset{\mathrm{def}}{=} \begin{cases} \mathit{true} & \text{if } \overline{\eta}[\![E_1]\!] = \overline{\eta}[\![E_2]\!] \\ \mathit{false} & \text{otherwise.} \end{cases} \tag{2.6}$$

We assume that the carrier set of `Bool` is as in Figure 2.1 in all algebras; hence *true* and *false* are objects in all algebras.

We also use the following shorthand for adding a binding to an assignment:

$$\eta[q/\mathbf{x}] \overset{\mathrm{def}}{=} \lambda l.\ \text{if } l = \mathbf{x} \text{ then } q \text{ else } \eta(l). \tag{2.7}$$

Definition 2.2.1 (satisfies for operations). Consider the following operation specification:

> **ndop** g($\vec{\mathbf{x}} : \vec{\mathrm{S}}$) **returns**(y:T)
>   **requires** P
>   **effect** Q

where and P and Q are boolean terms such that the free identifiers of P are from the $\mathbf{x}_i : \mathrm{S}_i$ and the free identifiers of Q are from the $\mathbf{x}_i : \mathrm{S}_i$ and $\mathrm{y} : \mathrm{T}$. Let X be the set of the $\mathbf{x}_i : \mathrm{S}_i$. The operation $A_{\mathrm{g}_{\vec{\mathrm{S}} \rightarrow \mathrm{T}}}$ *satisfies the specification of* g given above if and only if for all assignments η whose domain is X and whose range is A_{TYPES}, the following condition holds. If

$$\overline{\eta}[\![P]\!] \;=\; \mathit{true}, \tag{2.8}$$

then for all possible results $q \in A_{\mathrm{g}_{\vec{\mathrm{S}} \rightarrow \mathrm{T}}}(\eta(\vec{\mathbf{x}}))$,

$$q \;\neq\; \bot \tag{2.9}$$

$$\overline{\eta[q/\mathrm{y}]}[\![Q]\!] \;=\; \mathit{true}. \tag{2.10}$$

Furthermore, whenever some argument to the operation is $\perp$, then the only possible result is $\perp$.

We say a tuple of proper arguments *satisfies the requires clause* "**requires** P" if when η is an assignment that maps the formals to the given tuple of arguments, then $\overline{\eta}[\![P]\!] = \mathit{true}$.

If g is specified with **op** instead of **ndop** then an operation $A_{\mathtt{g}_{\vec{\mathtt{S}}\to\mathtt{T}}}$ satisfies the specification of g with nominal signature $\vec{\mathtt{S}} \to \mathtt{T}$ if in addition to the above requirements $A_{\mathtt{g}_{\vec{\mathtt{S}}\to\mathtt{T}}}$ has only one possible result on all proper arguments $\vec{q}$ that satisfy the requires clause of g.

For example, the operation defined by

$$A_{\mathtt{first}_{\mathtt{IntPair}\to\mathtt{Int}}}(p) \overset{\text{def}}{=} \{A_{\#.\mathrm{first}}(p)\} \tag{2.11}$$

$$A_{\mathtt{first}_{\mathtt{IntPair}\to\mathtt{Int}}}(\perp) \overset{\text{def}}{=} \{\perp\} \tag{2.12}$$

(where p is proper) satisfies the specification of the `first` operation of type `IntPair` in Figure 2.2, because for all assignments η that bind some proper IntPair p to p, $\overline{\eta}[\![\mathit{true}]\!] = \mathit{true}$, and the only possible result is $A_{\#.\mathrm{first}}(p)$. This result is proper and such that

$$\overline{\eta[A_{\#.\mathrm{first}}(p)/\mathrm{i}]}[\![\mathrm{i} = \mathrm{p.first}]\!] = \mathit{true}. \tag{2.13}$$

(Recall that an omitted requires clause is syntactic sugar for "**requires** true.")

The nullary operations that name class objects are implicitly specified as follows:

op T() **returns**(y:TClass)
  effect y = T.

An operation $A_{\mathtt{T}_{\to\mathtt{TClass}}}$ satisfies this specification if its only possible result is the result of $A_{\mathrm{T}}()$, which invokes the trait function T.

Our model of the visible types is the algebra that combines Figures 2.1 (`Bool`), B.1 (`Int`), and B.2 (`IntStream`) and a model of `BoolStream` that is analogous to `IntStream`. These are discussed in Appendix B.

We can now give a formal definition of when an algebra satisfies a specification.

Definition 2.2.2 (satisfies). An algebra A *satisfies* a specification *SPEC* if and only if

- $SIG(A) = SIG(SPEC)$,
- the functional structure of A satisfies the traits of $SPEC$,
- for each operation symbol $\mathbf{g}_{\vec{\mathtt{S}} \to \mathtt{T}}$ in $SIG(A)$, $A_{\mathbf{g}_{\vec{\mathtt{S}} \to \mathtt{T}}}$ satisfies the specification of $\mathbf{g}$ with nominal signature $\vec{\mathtt{S}} \to \mathtt{T}$, and
- the $SIG(B)$-reduct of A is B, where B is the algebra described above that models the visible types.

For example, the algebra A of Figure 2.5 is an IPT-algebra; that is, A satisfies the specification IPT.

2.3 Specifying Nondeterministic Types

Nondeterministic operations are useful for modeling both "undefined" behavior and types that are inherently nondeterministic. An example is the type `Mob` specified in Figure 2.6.

The **next** operation of `Mob` has a non-trivial precondition, specified in its **requires** clause. When the requires clause is not satisfied, the set of possible results can be the entire carrier set of the nominal return type. That is, we can have a Mob-algebra, B, whose carrier set for `Mob` is the set of finite sets of integers (plus $\perp$), and in which

$$B_{\mathbf{next}_{\mathtt{Mob} \to \mathtt{Int}}}(\{\}) \overset{\text{def}}{=} B_{\mathtt{Int}} = \{\perp, 0, 1, -1, \ldots\}. \tag{2.14}$$

However, we can also have Mob-algebras in which **next** is more deterministic (i.e., more "defined"). For example, we could have a Mob-algebra B' where

$$B'_{\mathbf{next}_{\mathtt{Mob} \to \mathtt{Int}}}(\{\}) \overset{\text{def}}{=} \{0\}. \tag{2.15}$$

To allow an operation to be nondeterministic on arguments that satisfy its requires clause, we must write **ndop** instead of **op** in the operation's specification. For example, if B is the Mob-algebra described above, then we can let

$$B_{\mathbf{next}_{\mathtt{Mob} \to \mathtt{Int}}}(m) \overset{\text{def}}{=} m \tag{2.16}$$

for all nonempty finite sets of integers m. Thinking of operations as relations, the operation $B_{\mathbf{next}_{\mathtt{Mob} \to \mathtt{Int}}}$ is the largest relation such that the specification is satisfied; the only limitation is that each possible result must be an element of the argument set.

Figure 2.5: An IPT-algebra, A.

Carrier Sets

$$
\begin{array}{lll}
A_{\texttt{IntPair}} & \overset{\text{def}}{=} & \{\bot\} \cup \{\langle i,j\rangle \mid (i,j \in A_{\texttt{Int}}) \wedge (i,j \neq \bot)\} \\
A_{\texttt{IntPairClass}} & \overset{\text{def}}{=} & \{\bot, \mathit{IntPair}\} \\
A_{\texttt{IntTriple}} & \overset{\text{def}}{=} & \{\bot\} \cup \{\langle i,j,k\rangle \mid (i,j,k \in A_{\texttt{Int}}) \wedge (i,j,k \neq \bot)\} \\
A_{\texttt{IntTripleClass}} & \overset{\text{def}}{=} & \{\bot, \mathit{IntTriple}\} \\
A_{\texttt{Bool}} & \overset{\text{def}}{=} & \{\bot, \mathit{true}, \mathit{false}\} \\
\vdots & &
\end{array}
$$

Trait Functions

$$
\begin{array}{lll}
A_{\text{IntPair}}() & \overset{\text{def}}{=} & \mathit{IntPair} \\
A_{(\#,\#)}(i,j) & \overset{\text{def}}{=} & \langle i,j\rangle \\
A_{\#.\text{first}}(\langle i,j\rangle) & \overset{\text{def}}{=} & i \\
A_{\#.\text{second}}(\langle i,j\rangle) & \overset{\text{def}}{=} & j \\
A_{\text{IntTriple}}() & \overset{\text{def}}{=} & \mathit{IntTriple} \\
A_{(\#,\#,\#)}(i,j,k) & \overset{\text{def}}{=} & \langle i,j,k\rangle \\
A_{\#[1]}(\langle i,j,k\rangle) & \overset{\text{def}}{=} & i \\
A_{\#[2]}(\langle i,j,k\rangle) & \overset{\text{def}}{=} & j \\
A_{\#[3]}(\langle i,j,k\rangle) & \overset{\text{def}}{=} & k \\
A_{\text{Bool}}() & \overset{\text{def}}{=} & \mathit{Bool} \\
\vdots & &
\end{array}
$$

Operations

$$
\begin{array}{lll}
A_{\texttt{IntPair}_{\to\texttt{IntPairClass}}}() & \overset{\text{def}}{=} & \{\mathit{IntPair}\} \\
A_{\texttt{make}_{\texttt{IntPairClass,Int,Int}\to\texttt{IntPair}}}(\mathit{IntPair}, i, j) & \overset{\text{def}}{=} & \{\langle i,j\rangle\} \\
A_{\texttt{first}_{\texttt{IntPair}\to\texttt{Int}}}(p) & \overset{\text{def}}{=} & \{A_{\#.\text{first}}(p)\} \\
A_{\texttt{second}_{\texttt{IntPair}\to\texttt{Int}}}(p) & \overset{\text{def}}{=} & \{A_{\#.\text{second}}(p)\} \\
A_{\texttt{IntTriple}_{\to\texttt{IntTripleClass}}}() & \overset{\text{def}}{=} & \{\mathit{IntTriple}\} \\
A_{\texttt{make}_{\texttt{IntTripleClass,Int,Int,Int}\to\texttt{IntTriple}}}(\mathit{IntTriple}, i, j, k) & \overset{\text{def}}{=} & \{\langle i,j,k\rangle\} \\
A_{\texttt{first}_{\texttt{IntTriple}\to\texttt{Int}}}(t) & \overset{\text{def}}{=} & \{A_{\#[1]}(t)\} \\
A_{\texttt{second}_{\texttt{IntTriple}\to\texttt{Int}}}(t) & \overset{\text{def}}{=} & \{A_{\#[2]}(t)\} \\
A_{\texttt{third}_{\texttt{IntTriple}\to\texttt{Int}}}(t) & \overset{\text{def}}{=} & \{A_{\#[3]}(t)\}
\end{array}
$$

$A_{\texttt{Bool}_{\to\texttt{BoolClass}}}() \overset{\text{def}}{=} \{\mathit{Bool}\}$

$\vdots$

Figure 2.6: Specification of the nondeterministic scheduler type `Mob`.

```
Mob immutable type
    class ops [new] instance ops [ins, waiting?, empty? next]
    based on sort C from Set with [Int for T]

    op new(c:MobClass) returns(m:Mob)
        effect m = {}

    op ins(m:Mob, i:Int) returns(r:Mob)
        effect r = m ∪ {i}

    op waiting?(m:Mob, i:Int) returns(b:Bool)
        effect b = (i ∈ m)

    op empty?(m:Mob) returns(b:Bool)
        effect b = (m = {})

    ndop next(m:Mob) returns(i:Int)
        requires m ≠ {}
        effect i ∈ m
```

Again, an operation can satisfy the specification of `next` without having this much nondeterminism. For example, consider an algebra C that is the same as B above except that, for all nonempty finite sets of integers m,

$$C_{\texttt{next}_{\texttt{Mob}\to\texttt{Int}}}(m) \overset{\text{def}}{=} \{\min(m)\}. \tag{2.17}$$

2.4 Specifying Types with Exceptions

Instead of specifying operations with non-trivial preconditions or arbitrarily defining a result, we often wish an operation to signal an *exception* [Goo75]. A programming language can define a mechanism to handle exceptions that arise while executing an invocation, as is done in CLU [LS79] and Trellis/Owl. When we discuss exceptions, we suppose that all operations of an algebra return `OneOf` objects. A `OneOf` object with tag `normal` models the normal return, while a `OneOf` objects with other tags model exceptional results.

A `OneOf` type is like the variant or discriminated union types that appear in some

Figure 2.7: The trait OneOf[normal: Int, empty: Null].

```
OneOf[normal: Int, empty: Null]: trait
    introduces
        make_normal: Int → NE
        make_empty: Null → NE
        hasTag?: NE, Tag → Bool
        val_normal: NE → Int
        val_empty: NE → Null
    asserts for all [i: Int]
        hasTag?(make_normal(i), normal) = true
        hasTag?(make_empty(nil), empty) = true
        hasTag?(make_empty(nil), normal) = false
        hasTag?(make_normal(i), empty) = false
        val_normal(make_normal(i)) = i
        val_empty(make_empty(nil)) = nil
    exempts for all [i: Int]
        val_normal(make_empty(nil))
        val_empty(make_normal(i))
```

programming languages, such as CLU [LAB*81]. The carrier set and trait functions associated with `OneOf` types are defined as shown by the example trait "OneOf[normal: Int, empty: Null]," which is found in Figure 2.7. (The type `Null` has only one proper object, denoted by the result of the trait function "nil"; it is used as a placeholder in `OneOf` types. The type `Tag` contains proper objects for each possible `OneOf` tag.) The generic operation symbols for a `OneOf` type are similarly defined.

To explain our syntactic sugar for specifications that use exceptions, consider the specification of the type `Mob2` given in Figure 2.8.

We rewrite each operation specification that specifies an exception so that it returns a `OneOf` type instead. The "normal" return from an operation is denoted by a `OneOf` object with the tag `normal`. (We do not allow `normal` as an exception name.) Each exception result is denoted by a `OneOf` object with a tag that is the same as the exception's name. Therefore the specification of `next` in `Mob2` is syntactic sugar for the specification of Figure 2.9.

All `OneOf` types used in this fashion have their specification implicitly included in a specification that uses them.

Figure 2.8: Specification of the type **Mob2**.

```
Mob2 immutable type
    class ops [new] instance ops [ins, waiting?, empty?, next]
    based on sort C from Set with [Int for T]

    op new(c:Mob2Class) returns(m:Mob2)
        effect m = {}

    op ins(m:Mob2, i:Int) returns(r:Mob2)
        effect r = m ∪ {i}

    op waiting?(m:Mob2, i:Int) returns(b:Bool)
        effect b = (i ∈ m)

    op empty?(m:Mob2) returns(b:Bool)
        effect b = (m = {})

    ndop next(m:Mob2) returns(i:Int) signals(empty(Null))
        effect ((m = {}) ⇒ signals empty(nil(Null)))
            & ((m ≠ {}) ⇒ i ∈ m)
```

Figure 2.9: Desugared form of an exception specification.

```
ndop next(m:Mob2) returns(o: OneOf[normal: Int, empty: Null])
    effect ((m = {}) ⇒ o = make_empty(nil))
        & ((m ≠ {}) ⇒ (hasTag?(o,normal) & val_normal(o) ∈ m))
```

We can also specify an operation so that it has a choice between signalling and returning a normal result. (For example, see Figure 5.4 on page 95.)

Chapter 3

Observations by an Applicative Language

The purpose of this chapter is to give a formal definition of observations. Observations are used both to evaluate assertions (see Chapter 4) and in the definition of subtype relations (see Chapter 5). A set of observations characterizes a programming language. To make observations concrete and to provide a notation for examples, we describe an applicative programming language and how it determines a set of observations. Our language NOAL (Nondeterministic Object-oriented Applicative Language) is a functional language based on the lambda calculus. We also use NOAL for examples when we discuss program verification in Chapter 7.

In the first section of this chapter, we define observations. In the remaining sections, we describe NOAL and its semantics.

3.1 Observations

We are interested in observations with free identifiers. Such observations are important because they model procedures that take arguments (existing objects) and manipulate them with generic invocations. These observations also highlight the problem of reasoning about programs that exploit inclusion polymorphism based on imprecise information about the types of arguments. That is, while each formal argument has a type, we only know that the actual argument's type is some subtype of the formal argument's type. This imprecise type information is characterized by the types of the free identifiers of our observations.

A *set of typed identifiers* is a *TYPES*-indexed family of disjoint sets, where *TYPES* is a set of type symbols. If X is a set of typed identifiers, and if $\mathbf{x} \in X_{\mathtt{T}}$ for some $\mathtt{T} \in \mathit{TYPES}$, then we write $\mathbf{x}\colon \mathtt{T}$, and say that $\mathbf{x}$ has *nominal type* $\mathtt{T}$. If X and Y are sets of typed identifiers indexed by the same set and if for each type T, $X_{\mathtt{T}} \subseteq Y_{\mathtt{T}}$, then we write $X \subseteq Y$.

We use environments to give meaning to the free identifiers of an observation.

Definition 3.1.1 (environment). An *environment* is a mapping from a set of typed identifiers to the carrier set of an algebra.

We allow an environment to map an identifier $\mathbf{x}\colon \mathtt{T}$ to an object of the carrier set of an algebra, regardless of the object's type, since this happens in programs that exploit inclusion polymorphism. This is the major technical difference between our environments and the environments used in traditional semantics, where each identifier of a given nominal type is mapped to an element of that type's carrier set.

For convenience, we write $ENV(X, A)$ for the set of all environments whose domain is a set X of typed identifiers, indexed by a set $TYPES$ of type symbols, and whose range is the $TYPES$-indexed family of carrier sets A_{TYPES} of an algebra A. We often bind identifiers in environments using the following notation:

$$\eta[q/\mathbf{x}] \stackrel{\text{def}}{=} \lambda l.\ \text{if } l = \mathbf{x} \text{ then } q \text{ else } \eta(l). \tag{3.1}$$

Observations need both an environment and an algebra to produce a set of possible results.

Definition 3.1.2 (observation). Let X be a set of typed identifiers. An *observation with free identifiers from* X is a mapping that takes an algebra A and an environment from $ENV(Y, A)$, where $X \subseteq Y$, and returns a set of possible results that have visible type; that is, if $SIG(A) = (SORTS, TYPES, V, TFUNS, OPS)$, then each possible result has some type $\mathtt{T} \in V$.

We require that the possible results of an observation have visible type, because this allows us to use an observation on different algebras and compare the sets of possible results. (Recall that we assume that the carrier sets and trait functions associated with the visible types are the same in all algebras we observe.)

In the next section we define a programming language with a generic invocation mechanism that allows one to write observations.

3.2 The Programming Language NOAL

The rest of this chapter describes the language NOAL, which is a hybrid of Trellis/Owl [SCB*86] and Broy's AMPL [Bro86]. We have taken AMPL's lambda calculus and explicit facilities for nondeterminism and added to it Trellis/Owl's type system and generic invocation mechanism. Type information aids reasoning, but we give a meaning to programs regardless of whether they are type-safe. Nondeterminism is necessary to make certain observations on nondeterministic types [Nip86]. The NOAL erratic choice expression, $1 \, [] \, 2$, has both 1 and 2 as possible results. Operationally, the erratic choice operator picks one expression at random and evaluates it. Therefore, if evaluation of one expression may not halt, then the entire expression may not halt. An angelic choice expression of the form $E_1 \;\bigtriangledown\; E_2$ will always halt if either E_1 or E_2 always halts. Operationally, the angelic choice operator runs both expressions in parallel and returns the first result. NOAL is a first-order language; that is functions are not objects in NOAL programs.

NOAL is only half of an object-oriented language, since we do not provide a class mechanism in NOAL. Formally, NOAL programs manipulate the algebras described in the previous chapter. That is, the meaning of a NOAL program is an observation. The identifiers in a NOAL program denote elements of the carrier set of an algebra. However, a NOAL program cannot directly call the operations of an algebra; instead it can only make generic invocations. The set of possible results of a generic invocation is determined by consulting the appropriate operation of an algebra.

The semantics of NOAL programs makes certain mild assumptions about algebras. The exact conditions are described in the sections on generic invocation below and in the section on domains in Appendix C. These conditions are satisfied by the algebraic models of most specifications written in our specification language. However, NOAL is not restricted to observing models of our specifications.

The type system of NOAL uses a nominal signature map and a presumed subtype relation to do type checking. It assumes that the nominal signature map is like that generated by a specification written in our specification language.

In the rest of this chapter we describe NOAL's syntax, generic invocation mechanism, semantics, and type system.

Figure 3.1: Syntax of NOAL.

```
⟨program⟩ ::= ⟨expr⟩ | ⟨rec fun def⟩ ⟨program⟩

⟨rec fun def⟩ ::= fun ⟨fun identifier⟩ ( ⟨decl list⟩ ) :  ⟨type⟩ = ⟨expr⟩ ;

⟨decl list⟩ ::= ⟨decl⟩ | ⟨decl list⟩ , ⟨decl⟩

⟨decl⟩ ::= ⟨identifier⟩ :  ⟨type⟩

⟨expr⟩ ::= ⟨identifier⟩
    | bottom [ ⟨type⟩ ]
    | ⟨generic operation⟩ ( )
    | ⟨generic operation⟩ ( ⟨expr list⟩ )
    | ⟨fun identifier⟩ ( ⟨expr list⟩ )
    | ( ⟨function abstract⟩ ) ( ⟨expr⟩ )
    | if ⟨expr⟩ then ⟨expr⟩ else ⟨expr⟩ fi
    | ⟨expr⟩ ▯ ⟨expr⟩
    | ⟨expr⟩ ▽ ⟨expr⟩
    | isDef?  ( ⟨expr⟩ )
    | ( ⟨expr⟩ )

⟨function abstract⟩ ::= fun ( ⟨decl list⟩ ) ⟨expr⟩

⟨expr list⟩ ::= ⟨expr⟩ | ⟨expr list⟩ , ⟨expr⟩
```

3.3 NOAL Syntax

The syntax of NOAL is presented in Figure 3.1. The nonterminal ⟨type⟩ denotes a type symbol, and ⟨generic operation⟩ denotes a generic operation symbol. We do not further specify the syntax of identifiers or function identifiers. However, we do not allow identifiers or function identifiers appearing in a program to be the same as the generic operation symbols used in that program. We assume that `isDef?` is neither a function identifier nor a generic operation symbol.

We also use the following syntactic sugars. We adopt Broy's notation for streams by considering an expression of the form E_1 & E_2 to be syntactic sugar for `cons(`E_2`,`E_1`)`. A declaration such as `f,s: Int` is sugar for the declaration list `f: Int, s: Int`. A generic operation symbol such as `T` used without a list of arguments is sugar for `T()`. We use **`true`** and **`false`** as sugar for `true(Bool())` and `false(Bool())` and we use 1, 2, and so on as sugar for `one(Int())`, `add(one(Int()),one(Int()))`, and so on.

The following example program has the stream $\langle 4\rangle$ as its only possible result (when applied to an algebra that includes the types `Int` and `IntStream`).

```
fun sumFirst (p1, p2:IntPair):Int = add(first(p1),first(p2));
sumFirst(make(IntPair,1,2), make(IntTriple(3,4,5))) & empty(IntStream)
```

3.4 Generic Invocation in NOAL

In this section we explain how we translate generic invocations in NOAL into calls on the operations of an algebra whose signature is $\Sigma = (SORTS, TYPES, V, TFUNS, OPS)$.

The translation from a generic operation symbol to an operation symbol is done by a *generic invocation mapping*, written *Generic*. This mapping takes a generic operation symbol and the actual arguments of the invocation (not the expressions that denote the arguments), and returns an operation symbol from *OPS* or the special symbol `typeError`. It operates by subscripting the generic operation symbol with the types of the actual arguments, and then finding an operation symbol with the same actual argument types. For example, if A is the IPT-algebra of Figure 2.5, then

$$Generic(\texttt{first}, \langle 1,2,3\rangle) = \texttt{first}_{\texttt{IntTriple}\to\texttt{Int}}.$$

Since an operation can only be applied to arguments having the types declared in the algebra's signature, if there is no operation symbol that matches the actual argument types exactly, then *Generic* must return `typeError`.

Unlike the generic invocation mechanism of a programming language such as Smalltalk-80, the NOAL generic invocation mechanism has to find a nonpolymorphic operation. This limitation of our algebraic models imposes a restriction on the algebras that NOAL programs can observe. In Smalltalk-80, an operation may return objects of a type that depends on the values of the operation's arguments. Such a Smalltalk-80 operation would have to be modeled by several operations in one of our algebras, each named by an operation symbol whose return type corresponds to the type of object it can return. Therefore we assume that whenever there is more than one operation symbol with the same name except for the return type in its subscript (e.g., $\mathtt{g}_{\bar{\mathtt{S}}\to\mathtt{T1}}$ and $\mathtt{g}_{\bar{\mathtt{S}}\to\mathtt{T2}}$), then for each tuple of actual arguments, no more than one of these operations has proper (i.e., non-$\bot$) possible results. This assumption allows *Generic* to find the correct operation symbol from the list of actual arguments (or to pick arbitrarily if all are $\bot$). This problem also shows why *Generic* needs the actual arguments and not just the types of the actual arguments.

To handle nonstrict operations properly, such as the `cons` operation of `BoolStream` and `IntStream` (see Figure B.2), the generic invocation mapping cannot be strict. To handle `cons` we must have for $b \in \{true, false\}$ and for $i \in \{0, 1, -1, \ldots\}$:

$$Generic(\mathtt{cons}, \bot, b) \overset{\text{def}}{=} \mathtt{cons}_{\mathtt{BoolStream,Bool}\to\mathtt{BoolStream}} \tag{3.2}$$

$$Generic(\mathtt{cons}, \bot, i) \overset{\text{def}}{=} \mathtt{cons}_{\mathtt{IntStream,Int}\to\mathtt{IntStream}}. \tag{3.3}$$

We can have *Generic* depend on only some of its arguments like this, provided it always has enough information to make the necessary choices. Our type specification language only allows one to specify types with strict operations. Therefore, for algebraic models of specifications written in our specification language, the only *nonstrict operations* are the `cons` operations of the types `IntStream` and `BoolStream`.

3.5 NOAL Semantics

The meaning of a program is an observation, which is a mapping from an algebra-environment pair to a set of possible results. Throughout the following we fix a signature $\Sigma = (SORTS, TYPES, V, TFUNS, OPS)$ and a Σ-algebra A.

3.5.1 Semantics of NOAL Expressions

In this section we give the semantics of NOAL expressions.

The meaning of an expression is described by the function $\mathcal{M}$, which takes an expression with free identifiers and function identifiers from a set X and returns an internal observation with free identifiers and function identifiers from X. An *internal observation* is an observation whose results need not have visible type.

The following list gives the denotation of each recursion-free NOAL expression in an environment $\eta \in ENV(X, A)$. For convenience, we assume that η also maps typed function identifiers to the denotations of recursively-defined NOAL functions.

- The only possible result of an identifier is its value in the environment η.

$$\mathcal{M}[\![\mathtt{x}]\!](A,\eta) \overset{\text{def}}{=} \{\eta(\mathtt{x})\} \tag{3.4}$$

- The only possible result of an expression of the form `bottom [ ⟨type⟩ ]` is $\perp$. The type is only included in the expression to make type-checking easier. We have, for each type T,

$$\mathcal{M}[\![\mathtt{bottom[T]}]\!](A,\eta) \overset{\text{def}}{=} \{\perp\} \tag{3.5}$$

- The possible results of a nullary generic operation are determined by consulting the corresponding operation.

$$\mathcal{M}[\![\mathtt{T()}]\!](A,\eta) \overset{\text{def}}{=} \begin{cases} A_{(\mathit{Generic}(\mathtt{T}))}() & \text{if } \mathit{Generic}(\mathtt{T}) \neq \mathtt{typeError} \\ \{\perp\} & \text{otherwise} \end{cases} \tag{3.6}$$

- The possible results of a generic invocation are determined by consulting the algebra for each possible argument list.

$$\begin{aligned} &\mathcal{M}[\![\mathtt{g}(\vec{E})]\!](A,\eta) \\ &\overset{\text{def}}{=} \bigcup_{\vec{q}\in\mathcal{M}[\![\vec{E}]\!](A,\eta)} \begin{cases} A_{(\mathit{Generic}(\mathtt{g},\vec{q}))}(\vec{q}) & \text{if } \mathit{Generic}(\mathtt{g},\vec{q}) \neq \mathtt{typeError} \\ \{\perp\} & \text{otherwise} \end{cases} \end{aligned} \tag{3.7}$$

- The possible results of a recursively defined function invocation are determined similarly, except that the meaning of the function identifier is found in the environment η.

$$\mathcal{M}[\![\mathtt{f}(\vec{E})]\!](A,\eta) \overset{\text{def}}{=} \bigcup_{\vec{q}\in\mathcal{M}[\![\vec{E}]\!](A)(\eta)} (\eta(\mathtt{f}))(\vec{q}) \tag{3.8}$$

- The set of possible results of a combination is the set of possible results of the body of the function abstract in each environment that extends the original by binding a list of possible arguments to the formals.

$$\mathcal{M}[\![\mathtt{(fun\ (}\vec{\mathtt{x}}:\vec{\mathtt{S}}\mathtt{)}\ E_0\mathtt{)(}\vec{E}\mathtt{)}]\!](A,\eta) \overset{\text{def}}{=} \bigcup_{\vec{q}\in\mathcal{M}[\![\vec{E}]\!](A,\eta)} \mathcal{M}[\![E_0]\!](A,\eta[\vec{q}/\vec{\mathtt{x}}]) \tag{3.9}$$

- The set of possible results for an `if` expression depend on the possible results for the first subexpression.

$$\begin{aligned}&\mathcal{M}[\![\texttt{if } E_1 \texttt{ then } E_2 \texttt{ else } E_3 \texttt{ fi}]\!](A,\eta)\\ &\stackrel{\text{def}}{=} \bigcup_{q\in\mathcal{M}[\![E_1]\!](A,\eta)} \begin{cases}\mathcal{M}[\![E_2]\!](A,\eta) & \text{if } q = \textit{true}\\ \mathcal{M}[\![E_3]\!](A,\eta) & \text{if } q = \textit{false}\\ \{\bot\} & \text{otherwise}\end{cases}\end{aligned} \tag{3.10}$$

- The set of possible results of an erratic choice expression includes those of both subexpressions.

$$\mathcal{M}[\![E_1 \,[\!]\, E_2]\!](A,\eta) \stackrel{\text{def}}{=} \mathcal{M}[\![E_1]\!](A,\eta) \cup \mathcal{M}[\![E_2]\!](A,\eta) \tag{3.11}$$

- The set of possible results of an angelic choice expression is the same as an erratic choice, except that $\bot$ is a possible result only if it is a possible result of both subexpressions.

$$\begin{aligned}&\mathcal{M}[\![E_1 \triangledown E_2]\!](A,\eta)\\ &\stackrel{\text{def}}{=} \begin{cases}\mathcal{M}[\![E_1]\!](A,\eta) \cup (\mathcal{M}[\![E_2]\!](A,\eta) \setminus \{\bot\}) & \text{if } \bot \notin \mathcal{M}[\![E_1]\!](A,\eta)\\ (\mathcal{M}[\![E_1]\!](A,\eta) \setminus \{\bot\}) \cup \mathcal{M}[\![E_2]\!](A,\eta) & \text{if } \bot \notin \mathcal{M}[\![E_2]\!](A,\eta)\\ \mathcal{M}[\![E_1]\!](A,\eta) \cup \mathcal{M}[\![E_2]\!](A,\eta) & \text{otherwise}\end{cases}\end{aligned} \tag{3.12}$$

  The set $s_1 \setminus s_2$, consists of the elements of s_1 that are not elements of s_2.

- The primitive `isDef?` can be used to see if an expression's possible results are proper.

$$\mathcal{M}[\![\texttt{isDef?}(E)]\!](A,\eta) \stackrel{\text{def}}{=} \bigcup_{q\in\mathcal{M}[\![E]\!](A,\eta)} \begin{cases}\{\textit{true}\} & \text{if } q \neq \bot\\ \{\bot\} & \text{otherwise}\end{cases} \tag{3.13}$$

  Since there is no "assignment statement" the only way to bind objects to identifiers is through function calls or the application of a function abstract. The parameter passing mechanism of NOAL is lazy evaluation, so `isDef?` is needed to define strict functions.

3.5.2 Semantics of Recursive Function Definitions

NOAL programs may begin with a system of mutually recursive function definitions. In the body of a recursively defined function, there can be no free identifiers or function identifiers, besides those of the other recursively defined functions and the function's formal arguments.

Since NOAL "functions" can be nondeterministic, their denotations are operations. Recall that an operation is a mapping that takes a tuple of zero or more elements of a carrier set and returns a nonempty set of possible results.

It is a difficult problem to find the least fixed points of systems of mutually recursive function definitions in NOAL. We discuss the construction of least fixed points in Appendix C. In this section we merely give some informal explanations and examples.

NOAL uses lazy evaluation for evaluating function arguments [Sch86, Page 181]; Broy calls the rule *call-time-choice*. Like call-by-name, call-time-choice uses delayed evaluation, hence functions written in NOAL need not be strict. However, each actual parameter is only evaluated once; hence formal parameters are not themselves sources of nondeterminism. That is, if a formal argument is mentioned twice in the body of a lambda-abstraction, the same value will be substituted in each instance. The following program demonstrates the difference between call-time-choice and call-by-name:

```
fun f (x:Int):  Int = add(x,x); f(0 [] 1).
```

In our call-time-choice semantics this program has as possible results both 0 and 2; a result of 1 is not possible with call-time-choice, although it would be possible with call-by-name. Another interesting example is the following program:

```
fun choose(x:Int):  Int = x ∇ choose(add(x,1)); choose(0).
```

This program has as its set of possible results all positive integers; furthermore, it is guaranteed to terminate! If we replaced the angelic choice (∇) with an erratic choice ($[]$) in this program, the program would also have all positive integers as possible results, but in addition it might not terminate.

3.5.3 Semantics of NOAL Programs

We also use $\mathcal{M}$ to denote the function that takes a program with free identifiers from some set of typed identifiers, and returns an observation with free identifiers from the same set. Consider the program $\vec{F}; E$, with a list of recursive function definitions $\vec{F}$, an expression E, and whose free identifiers are taken from a set X. Let $\eta \in ENV(Y, A)$ be an environment, where $X \subseteq Y$. Let η' be $\eta[\vec{f}/\vec{\mathtt{f}}]$, that is, η extended by binding fixed points $\vec{f}$ to the function identifiers $\vec{\mathtt{f}}$ defined in $\vec{F}$. We ensure that all possible results of a program have visible type using the following function:

$$Visible(q) \overset{\text{def}}{=} \begin{cases} q & \text{if } q \in A_{\mathtt{T}} \text{ and } \mathtt{T} \in V \\ \bot & \text{otherwise} \end{cases} \tag{3.14}$$

We use the above conventions and the meaning of the expression E to define the meaning of the program $\vec{F}; E$ as follows:

$$\mathcal{M}[\![\vec{F}; E]\!](A, \eta) \overset{\text{def}}{=} Visible(\mathcal{M}[\![E]\!](A, \eta')). \tag{3.15}$$

3.6 Nominal Types and Type Checking for NOAL

Type-safe programs are interesting because they can only observe the results of an expression by invoking the instance operations of the expression's nominal type. For example, a type-safe program cannot apply `third` to an expression of nominal type `IntPair`. So when observed by a type-safe program, instances of `IntTriple` that are bound to identifiers of nominal type `IntPair` behave like instances of `IntPair`.

We describe type-safe programs and their nominal types by using a nominal signature map *NomSig*, and a presumed subtype relation $\leq$.

To support inclusion polymorphism, if $\mathtt{S} \leq \mathtt{T}$, then argument expressions of nominal type `S` can be used where arguments of nominal type `T` are expected [SCB*86].

For our purposes in Chapter 6, we want to ensure that each expression and program has only one nominal type. Therefore, we assume that the nominal signature map *NomSig* satisfies the following conditions, for all generic operation symbols `g` in the domain of *NomSig*.

- If $\rightarrow \mathtt{T} \in NomSig(\mathtt{g})$, then there is no other nominal signature in $NomSig(\mathtt{g})$ that has no argument positions. That is, there is at most one nominal signature of the form $\rightarrow \mathtt{T}$ in $NomSig(\mathtt{g})$.

- If $\vec{\mathtt{S}} \rightarrow \mathtt{T} \in NomSig(\mathtt{g})$ and $\vec{\mathtt{S}}$ is not empty, then every other element of $NomSig(\mathtt{g})$ with the same number of argument positions differs from $\vec{\mathtt{S}} \rightarrow \mathtt{T}$ in at least the first argument position.

While this is not the only way to ensure that each generic invocation expression has only one nominal type, it matches the nominal signature maps our specification language generates. Bruce and Wegner ensure that each expression has a single type by imposing a "regularity" condition on $\leq$ [BW87]. In Cardelli's type systems (e.g., [Car84]), each expression has several types.

Figure 3.2 shows the type inference rules for NOAL. These rules precisely define the *nominal type* of each NOAL expression. In the figure, H is a set of type assumptions of the form $\mathtt{x} : \mathtt{T}$, meaning that the identifier $\mathtt{x}$ has nominal type $\mathtt{T}$, or $\mathtt{f} : \vec{\mathtt{S}} \rightarrow \mathtt{T}$, meaning that the function identifier $\mathtt{f}$ has nominal signature $\vec{\mathtt{S}} \rightarrow \mathtt{T}$. We also use vector notation; $\vec{\mathtt{x}} : \vec{\mathtt{S}}$ means that each $\mathtt{x}_i$ has nominal type $\mathtt{S}_i$. The notation $H.\mathtt{x} : \mathtt{T}$ means H extended with the assumption $\mathtt{x} : \mathtt{T}$ (where the extension replaces all assumptions about $\mathtt{x}$ in H). The notation $H \vdash E : \mathtt{T}$ means that given H one can prove, using the inference rules, that the expression E has nominal type $\mathtt{T}$. An inference rule of the form:

$$\frac{h_1, h_2}{c}$$

means that to prove the conclusion c one must first show that both hypotheses h_1 and h_2 hold. Rules written without hypotheses and the horizontal line are axioms.

The only rules that allow one to exploit the presumed subtype relation $\leq$ are [ginvoc], [fcall] and [comb]. The rules [fcall] and [comb] allow the nominal type of an actual argument expression to be a presumed subtype of the formal's type. The rule [ginvoc] is similar, except that one must pick a nominal signature for the generic operation such that the nominal type of the first actual argument is the same as the nominal type of the first formal argument.

We do type checking on a program by using the nominal types of the program's free identifiers. Let X be a set of typed identifiers indexed by *TYPES*; that is, for each $\mathtt{x} \in X$, we associate a nominal type $\mathtt{T} \in$ *TYPES*. We can therefore regard X as a set of type assumptions. We say a program P whose set of free identifiers is X has nominal type $\mathtt{T}$ if we have $X \vdash P : \mathtt{T}$. Since the syntax of NOAL does not provide a way to declare the nominal type of each free identifier in a program, we always give this as auxiliary information.

The *set of type-safe expressions over NomSig and* $\leq$ is the set of all NOAL expressions

Figure 3.2: Type Inference Rules for NOAL

[ident] $H.\mathtt{x} : \mathtt{T} \vdash \mathtt{x} : \mathtt{T}$

[fident] $H.\mathtt{f} : \vec{\mathtt{S}} \rightarrow \mathtt{T} \vdash \mathtt{f} : \vec{\mathtt{S}} \rightarrow \mathtt{T}$

[ngop] $$\frac{\rightarrow \mathtt{TClass} \in \mathit{NomSig}(\mathtt{T})}{H \vdash \mathtt{T()} : \mathtt{TClass}}$$

[ginvoc] $$\frac{\vec{\mathtt{S}} \rightarrow \mathtt{T} \in \mathit{NomSig}(\mathtt{g}),\ H \vdash \vec{E} : \vec{\sigma},\ \sigma_1 = \mathtt{S}_1,\ \sigma_2 \leq \mathtt{S}_2, \ldots, \sigma_n \leq \mathtt{S}_\mathtt{n}}{H \vdash \mathtt{g}(\vec{E}) : \mathtt{T}}$$

[fcall] $$\frac{H \vdash \mathtt{f} : \vec{\mathtt{S}} \rightarrow \mathtt{T},\ H \vdash \vec{E} : \vec{\sigma},\ \vec{\sigma} \leq \vec{\mathtt{S}}}{H \vdash \mathtt{f}(\vec{E}) : \mathtt{T}}$$

[comb] $$\frac{H.\vec{\mathtt{x}} : \vec{\mathtt{S}} \vdash E_0 : \mathtt{T}_0,\ H \vdash \vec{E} : \vec{\sigma},\ \vec{\sigma} \leq \vec{\mathtt{S}}}{H \vdash \mathtt{(fun\ (}\vec{\mathtt{x}} : \vec{\mathtt{S}}\mathtt{)}\ E_0\mathtt{)}\ \mathtt{(}\vec{E}\mathtt{)} : \mathtt{T}_0}$$

[if] $$\frac{H \vdash E_1 : \mathtt{Bool},\ H \vdash E_2 : \mathtt{T},\ H \vdash E_3 : \mathtt{T}}{H \vdash \mathtt{(if}\ E_1\ \mathtt{then}\ E_2\ \mathtt{else}\ E_3\ \mathtt{fi)} : \mathtt{T}}$$

[erratic] $$\frac{H \vdash E_1 : \mathtt{T},\ H \vdash E_2 : \mathtt{T}}{H \vdash (E_1 \mathbin{[\,]} E_2) : \mathtt{T}}$$

[angelic] $$\frac{H \vdash E_1 : \mathtt{T},\ H \vdash E_2 : \mathtt{T}}{H \vdash (E_1 \bigtriangledown E_2) : \mathtt{T}}$$

[isDef] $H \vdash \mathtt{isDef?}(E) : \mathtt{Bool}$

[bot] $H \vdash \mathtt{bottom[T]} : \mathtt{T}$

[prog] $$\frac{\begin{array}{c} \mathtt{f}_1 : \vec{\mathtt{S}_1} \rightarrow \mathtt{T}_1, \ldots, \mathtt{f}_m : \vec{\mathtt{S}_m} \rightarrow \mathtt{T}_m, \vec{\mathtt{x}_1} : \vec{\mathtt{S}_1} \vdash E_1 : \mathtt{T}_1, \\ \vdots \\ \mathtt{f}_1 : \vec{\mathtt{S}_1} \rightarrow \mathtt{T}_1, \ldots, \mathtt{f}_m : \vec{\mathtt{S}_m} \rightarrow \mathtt{T}_m, \vec{\mathtt{x}_m} : \vec{\mathtt{S}_m} \vdash E_m : \mathtt{T}_m, \\ H.\mathtt{f}_1 : \vec{\mathtt{S}_1} \rightarrow \mathtt{T}_1, \ldots, \mathtt{f}_m : \vec{\mathtt{S}_m} \rightarrow \mathtt{T}_m \vdash E : \mathtt{T} \end{array}}{H \vdash \left(\begin{array}{l} \mathtt{fun}\ \mathtt{f}_1\ \ (\vec{\mathtt{x}}_1 : \vec{\mathtt{S}}_1)\mathtt{:T}_1\ \mathtt{=}\ E_1\mathtt{;} \\ \vdots \\ \mathtt{fun}\ \mathtt{f}_m\ \ (\vec{\mathtt{x}}_m : \vec{\mathtt{S}}_m)\mathtt{:T}_m\ \mathtt{=}\ E_1\mathtt{;} \\ E \end{array} \right) : \mathtt{T}}$$

that have a nominal type, when type-checked using *NomSig*, $\leq$ and the types of their free identifiers. The ***set of type-safe programs over NomSig and*** $\leq$ is the set of all NOAL programs whose nominal type is a visible type. If *SPEC* is a specification with nominal signature map *NomSig* and presumed subtype relation $\leq$, we write "the set of type-safe NOAL programs over *SPEC*."

The nominal type of an expression can be thought of as an upper bound on the types of values the expression can denote. That is, suppose E is a type-safe expression over *NomSig* and $\leq$. Then the elements of $\mathcal{M}[\![E]\!](A, \eta)$ must be instances of a type that is related by $\leq$ to E's nominal type, if $\leq$, *NomSig*, A, and η meet the following conditions. First, $\leq$ must be reflexive and transitive. Second, if $\mathtt{S} \leq \mathtt{T}$, then S must have all the instance operations of T, and the nominal signatures of these operations must be suitably related. This condition is formalized in the definition of safe relations below. Third, η must be such that every identifier is mapped to an instance of some type that is related by $\leq$ to the identifier's nominal type. This condition is formalized in the definition of an environment that obeys a relation below. Finally, the signature of A must be that of a specification with nominal signature map *NomSig* and presumed subtype relation $\leq$. In what follows, we formalize the above remarks.

The definition of safe relations parallels the syntactic restrictions placed on the subtype relations in Trellis/Owl [SCB*86].

Definition 3.6.1 (safe). Let *NomSig* be a nominal signature map. Let *GOP* be the domain of *NomSig*. Let $\leq$ be a binary relation on type symbols. Then $\leq$ is *safe with respect to NomSig* if and only if for every generic operation symbol $\mathtt{g} \in GOP$ the following property holds. If $\mathtt{T}, \mathtt{ArgT}_2, \ldots, \mathtt{ArgT}_n \rightarrow \mathtt{RetT}$ is an element of *NomSig*(g), and $\mathtt{S} \leq \mathtt{T}$, then there must be some a nominal signature $\mathtt{S}, \mathtt{ArgS}_2, \ldots, \mathtt{ArgS}_n \rightarrow \mathtt{RetS}$ in *NomSig*(g) such that $\mathtt{RetS} \leq \mathtt{RetT}$ and for all i from 2 to n, $\mathtt{ArgT}_i \leq \mathtt{ArgS}_i$.

For example, a relation such that $\mathtt{IntTriple} \leq \mathtt{IntPair}$ is safe with respect to the nominal signature map of IPT. However, a relation such that $\mathtt{IntPair} \leq \mathtt{IntTriple}$ is not safe.

Environments that obey a subtype relation are fundamental to our methods for specification and verification, as well as our definition of subtype relations.

Definition 3.6.2 (obeys). Let $\Sigma = (SORTS, TYPES, V, TFUNS, OPS)$ be a signature. Let X be a set of typed identifiers, indexed by $TYPES$. Let A be a Σ-algebra. Let $\leq$ be a binary relation on $TYPES$. Let $\eta \in ENV(X, A)$ be an environment. Then η *obeys* $\leq$ if and only if for every type $\mathtt{T} \in TYPES$ and for every $\mathtt{x}$ of nominal type $\mathtt{T}$ in X, $\eta(\mathtt{x})$ has a type $\mathtt{S}$ such that $\mathtt{S} \leq \mathtt{T}$.

Notice that no environment obeys the empty relation on types.

In the following lemma, we show that, if, in addition to the conditions listed above, each recursively defined function is such that whenever it is given arguments whose types are subtypes of the formal argument's nominal types, the type of each possible result is a subtype of the function's nominal result type, then the nominal type of an expression is an upper bound on the types of the expression's possible results.

This lemma is also the source of the conditions on $\leq$ listed above. The condition that $\leq$ be reflexive comes from expressions such as $\mathtt{T}$, where $\mathtt{T}$ is a type symbol, since the possible results of such an expression have the expression's nominal type. The condition that $\leq$ be transitive comes from function calls, where the nominal type of a formal may be $\mathtt{S}$, the nominal type of the actual may be $\sigma \leq \mathtt{S}$, and the type of the actual argument may be $\sigma' \leq \sigma$. The condition that $\leq$ be safe with respect to the nominal signature map comes from generic invocations, where the generic invocation's nominal signature must be appropriately related to the signature of the operation that the NOAL generic invocation mechanism selects.

To handle environments defined on function identifiers in the following lemma we define "obeys" for functions. Let A be an algebra and let η be such an environment whose range is A. We say that η *obeys* $\leq$ if for each $\mathtt{f} : \vec{\mathtt{S}} \rightarrow \mathtt{T}$ in the domain of η, whenever $\vec{\sigma} \leq \vec{\mathtt{S}}$ and $\vec{q} \in A_{\vec{\sigma}}$, then each possible result of $\eta(\mathtt{f})(\vec{q})$ has some type $\tau \leq \mathtt{T}$.

Lemma 3.6.3. Let $SPEC$ be a specification. Let $NomSig$ be the nominal signature map of $SPEC$. Let $\leq$ be the presumed subtype relation of $SPEC$. Let E be a NOAL expression of nominal type $\mathtt{T}$ whose set of free identifiers and free function identifiers is X. Let A be an algebra whose signature is $SIG(SPEC)$. Let $\eta \in ENV(X, A)$ be an environment.

Suppose $\leq$ is reflexive and transitive, $\leq$ is safe with respect to $NomSig$, and η obeys

$\leq$. Then each possible result of $\mathcal{M}[\![E]\!](A,\eta)$ has a type $\tau \leq \mathtt{T}$.

Proof: (by induction on the structure of expressions.)

As a basis, we show that the result holds for identifiers, `bottom[T]`, and nullary generic operations. If the expression is an identifier $\mathtt{x} : \mathtt{T}$, then $\mathcal{M}[\![\mathtt{x}]\!](A,\eta) = \{\eta(\mathtt{x})\}$ and by hypothesis, the type of $\eta(\mathtt{x})$ is related by $\leq$ to T. The result is trivial for `bottom[T]`, since the only possible result is $\perp$. The result is also trivial for a nullary generic operation symbol of type T, since $\leq$ is reflexive and the only possible result has type T.

For the inductive step we assume that the result holds for each subexpression.

- Suppose the expression is a generic invocation $\mathtt{g}(\vec{E})$. By the type inference rule [ginvoc], g has some nominal signature $\vec{\mathtt{S}} \rightarrow \mathtt{T}$, $\vec{E} : \vec{\sigma}$, $\sigma_1 = \mathtt{S}_1$, and for each i from 2 to n, $\sigma_i \leq \mathtt{S}_i$. Let $\vec{q}$ be a tuple of possible results from $\vec{E}$. By the inductive hypothesis, $\vec{q}$ has a type $\vec{\sigma'}$ such that $\vec{\sigma'} \leq \vec{\sigma}$. Since $\leq$ is safe with respect to *NomSig* and $\sigma'_1 \leq \mathtt{S}_1$, there is some nominal signature $\sigma'_1, \mathtt{ArgS}_2, \ldots, \mathtt{ArgS}_n \rightarrow \mathtt{RetS}$ in *NomSig*(g) such that $\mathtt{RetS} \leq \mathtt{T}$ and for all i from 2 to n, $\mathtt{S}_i \leq \mathtt{ArgS}_i$. By definition of *SIG*(*SPEC*), if for each i from 2 to n, $\sigma'_i = \mathtt{ArgS}_i$, then there is an operation symbol $\mathtt{g}_{\vec{\sigma'} \rightarrow \mathtt{RetS}}$ and so by the semantics of NOAL, each possible result has type $\mathtt{RetS} \leq \mathtt{T}$. If there is no such operation symbol, then the only possible result is $\perp$ which has type T.

- Suppose the expression is a function call $\mathtt{f}(\vec{E})$. By the type inference rule [fcall], f has some nominal signature $\vec{\mathtt{S}} \rightarrow \mathtt{T}$, $\vec{E} : \vec{\sigma}$, and $\vec{\sigma} \leq \vec{\mathtt{S}}$. Let $\vec{q}$ be a tuple of possible results from $\vec{E}$. By the inductive hypothesis, $\vec{q}$ has a type $\vec{\sigma'}$ such that $\vec{\sigma'} \leq \vec{\sigma}$. By hypothesis $\leq$ is transitive, so $\vec{\sigma'} \leq \vec{\mathtt{S}}$. Since η obeys $\leq$, each possible result has some type $\tau \leq \mathtt{T}$.

- Suppose the expression is a combination $(\mathtt{fun}\ (\vec{\mathtt{x}} : \vec{\mathtt{S}})\ E_0)\ (\vec{E})$. By the type inference rule [comb], E_0 has nominal type T, $\vec{E} : \vec{\sigma}$, and $\vec{\sigma} \leq \vec{\mathtt{S}}$. Let $\vec{q}$ be a tuple of possible results from $\vec{E}$. By the inductive hypothesis, $\vec{q}$ has a type $\vec{\sigma'}$ such that $\vec{\sigma'} \leq \vec{\sigma}$. By hypothesis $\leq$ is transitive, so $\vec{\sigma'} \leq \vec{\mathtt{S}}$. So the environment $\eta[\vec{q}/\vec{\mathtt{x}}]$ obeys $\leq$ and thus the result follows from the inductive hypothesis applied to E_0.

- The result follows directly from the inductive hypothesis for the other expressions.

∎

The following lemma says that the possible results of a recursively defined function in NOAL are related by $\leq$ to the function's nominal result type, provided the conditions above are met.

Lemma 3.6.4. Let *SPEC* be a specification. Let *NomSig* be the nominal signature map of *SPEC*. Let $\leq$ be the presumed subtype relation of *SPEC*. Let A be an algebra whose signature is $SIG(SPEC)$. Let $\vec{F}$ be a type-safe system of mutually recursive NOAL function definitions. Let f be a function in $\vec{F}$, the formals of f be $\vec{\mathtt{x}} : \vec{\mathtt{S}}$, the body of f be the expression E, and let f have nominal signature $\vec{\mathtt{S}} \rightarrow \mathtt{T}$. Let η be an environment whose domain is the $\mathbf{x}_i$ and whose range is A.

Suppose $\leq$ is reflexive and transitive, and $\leq$ is safe with respect to *NomSig*, and η obeys $\leq$. Then each possible result of $\mathcal{M}[\![E]\!](A,\eta)$ has a type $\tau \leq \mathtt{T}$.

Proof: Let $q \in \mathcal{M}[\![E]\!](A,\eta)$ be a possible result. If $q = \perp$ the result is trivial, so suppose $q \neq \perp$. Pick a computation that produces q. Since $q \neq \perp$, the computation uses only finitely many calls, say n, to the $\mathtt{f}_i$. Expand E by replacing each call to a function $\mathtt{f}_i \in \vec{F}$ with its body and repeating this process on the resulting expression n times and then replace all the remaining function calls with the expression bottom[S], where S is the nominal result type of the replaced call. This process does not change the nominal type of the resulting expression, and since there are no free function identifiers that remain, the result follows from the previous lemma. ∎

Since each recursively defined function preserves our view of types as an upper bound, each expression also preserves that view. So the following lemma differs from Lemma 3.6.3 in that no assumption is made about the environment's function identifiers.

Lemma 3.6.5. Let *SPEC* be a specification. Let *NomSig* be the nominal signature map of *SPEC*. Let $\leq$ be the presumed subtype relation of *SPEC*. Let E be a NOAL expression of nominal type T whose set of free identifiers is X. Let A be an algebra whose signature is $SIG(SPEC)$. Let $\eta \in ENV(X,A)$ be an environment.

Suppose $\leq$ is reflexive and transitive, $\leq$ is safe with respect to *NomSig*, and η obeys $\leq$. Then each possible result of $\mathcal{M}[\![E]\!](A,\eta)$ has a type $\tau \leq \mathtt{T}$.

Proof: By the previous lemma, if we extend η by binding the denotation of each recursively defined function identifier that is free in E to its denotation in A, then the extended environment obeys $\leq$. So by Lemma 3.6.3, the result follows. ∎

Chapter 4

Specifying Polymorphic Functions

In this chapter we describe a new method for the specification of functions that exploit inclusion polymorphism.

We write specifications as if each argument and result has the specified type, and then allow arguments and results to have types that are subtypes of the specified types. Our specifications follow the practice of Trellis/Owl and other languages that allow instances of subtypes to be used wherever instances of their supertypes can be used. An example is the specification of **`sumFirst`**, found in Figure 1.4 on page 15. It specifies that the arguments must be instances of a subtype of `IntPair`, but its effect is written using the trait functions for the type `IntPair`. The advantage of this approach is that the syntax and semantics of our specifications parallels that of the implementations. However, there is no standard semantics for such specifications when the actual arguments do not have the specified types.

We describe a semantics for such specifications. We know how to evaluate the assertions that specify the effect of a function in an environment where each formal argument identifier denotes an actual argument whose type is the same as the formal's nominal type. We call such an environment a *nominal environment.* To evaluate an assertion in an environment that is not nominal, one finds a nominal environment that the first environment imitates, and uses the nominal environment to evaluate the assertion. We define when one environment imitates another with respect to a set of observations, allowing the first environment to be more deterministic with respect to each observation in the set. We use this method for evaluating assertions to define satisfaction for NOAL function specifications.

Our semantics is independent of the definition of subtype (given in Chapter 5), since in this chapter we use binary relations on types without assumptions about what such a relation means.

After describing the syntax of function specifications, we define evaluation of assertions using the imitates relation and the semantics of polymorphic function specifications. We then discuss the limitations of our approach, the problems in adapting our approach to the specification of types with polymorphic operations, and why our semantics for function specifications is plausible.

4.1 Syntax of Function Specifications

To specify NOAL functions we use syntax similar to that for operation specifications, except that we use **fun** instead of **ndop**. That is, we give an operation name, a nominal signature, a ⟨requires clause⟩ and an ⟨effect clause⟩; the latter two contain terms written using trait function symbols (see Figure 2.4 for the syntax of terms).

Since an NOAL program is applicative, we also use this syntax to specify NOAL programs.

The trait functions used in a function specification must be taken from a specification of some abstract data types; we call the specification of these abstract types the *referenced specification*. The referenced specification is explicitly named in function specifications following the keyword **uses** (this is similar to Wing's specifications [Win83], although we are naming a specification of several abstract types instead of a trait).

For example, a specification we call `is2waiting` is given in Figure 4.1. The referenced specification is MP, which specifies the types `Mob` (see Figure 2.6 on page 39) and `PSchd` (see Figure 4.2).

The scheduler type `PSchd` (short for priority scheduler) is specified in Figure 4.2. (The trait OrderedSet that is defined below in Figure 4.3, extends the standard set trait

Figure 4.1: The function specification `is2waiting`.

```
fun is2waiting(m:Mob) returns(b:Bool)
    uses MP
    requires true
    effect b = (2 ∈ m)
```

Figure 4.2: Specification of the priority scheduler type, PSchd.

```
PSchd immutable type
    class ops [new] instance ops [ins, waiting?, next, empty?, leastFirst]
    based on sort C from Pair
        with [Bool for T1, OrderedSet with [Int for T] for T2]

    op new(c:PSchdClass, b:Bool) returns(p:PSchd)
        effect m = ⟨b, {}⟩

    op ins(p:PSchd, i:Int) returns(m:PSchd)
        effect m = ⟨p.first, p.second ∪ {i}⟩

    op waiting?(p:PSchd, i:Int) returns(b:Bool)
        effect b = i ∈ p.second

    op empty?(p:PSchd) returns(b:Bool)
        effect b = (p.second = {})

    op next(p:PSchd) returns(i:Int)
        requires p.second ≠ {}
        effect i ∈ p.second
                    & (p.first ⇒ lowerBound?(p.second,i))
                    & ((¬p.first) ⇒ upperBound?(p.second,i))
    op leastFirst(p:PSchd) returns(b:Bool)
        effect b = p.first
```

by adding the trait functions "lowerBound?" and "upperBound?".) The abstract values of `PSchd` instances can be thought of as pairs, consisting of a boolean that tells the priority order and an ordered set that contains the integers in the scheduler. The crucial difference from `Mob` is that the `next` operation of a priority scheduler must return either the least or the greatest integer waiting to be scheduled, with the priority determined by the boolean that is fixed when the object is created. The `leastFirst` operation returns the priority of a `PSchd` instance. If `leastFirst(q)` is true, then `next(q)` will return the least element of q.

The precondition of `is2waiting` is "true" and the postcondition is "$b = (2 \in m)$." Since many specifications have a precondition of "true" we consider an omitted requires clause to be syntactic sugar for a requires clause of the form "**requires** true."

Figure 4.3: The trait OrderedSet.

```
OrderedSet trait
    imports Set
    assumes Ordered
    introduces
        lowerBound?: C, T → Bool
        upperBound?: C, T → Bool
    asserts for all [s: C, i, j: T]
        lowerBound?({},i) = true
        lowerBound?(insert(s,j),i) = ((i ≤ j) & lowerBound?(s,i))
        upperBound?({},i) = true
        upperBound?(insert(s,j),i) = ((i ≥ j) & upperBound?(s,i))
```

The free identifiers of a function specification's precondition must be drawn from the formal arguments of the function specification (e.g., `m` in the `is2waiting` example). The free identifiers of the postcondition must be drawn from both the formal arguments of the function specification and the formal result identifier (e.g., `b` in the `is2waiting` example).

A term that sort-checks and only uses trait functions of the signature *SIG*(*SPEC*) of a type specification *SPEC* is a *SPEC-term*. For example, the assertions in the function specification `is2waiting` are MP-terms. A *SPEC*-term *sort checks* when the number and sorts of arguments to trait functions match the signatures specified in the traits of *SPEC*, and the only equations are between terms of the same sort. (An equation has sort `Bool`.) There is no generic invocation that applies to terms, so sort-checking for *SPEC*-terms is trivial. Therefore we can infer for each *SPEC*-term a nominal sort. Of particular interest are terms of nominal sort `Bool`.

Definition 4.1.1 (*SPEC*-assertion). A *SPEC-assertion* is a *SPEC*-term of nominal sort `Bool`.

4.2 Evaluation of Assertions

In this section we define the notion of when an algebra-environment pair models an assertion with respect to a set of observations. This is done by finding a nominal algebra-environment pair that the given pair imitates.

Formally, an environment η is *nominal* if and only if it obeys the identity relation on types $(=)$. We call an algebra-environment pair *nominal* if the environment is nominal.

In Chapter 2 we have described how to evaluate terms with free identifiers using a nominal environments, which in that chapter were called assignments. If $\eta_A \in ENV(X, A)$ is a nominal environment, then $\overline{\eta_A}$ is its extension to a mapping from terms to elements of the carrier set of A. Recall that $\overline{\eta_A}$ works by using the trait functions of A to evaluate trait function applications and using η_A itself to evaluate free identifiers.

4.2.1 The Imitates Relation

In this subsection we define when one algebra-environment pair imitates another with respect to a set of observations.

For soundness of reasoning, we want one algebra-environment pair to imitate another only if there is no observation that distinguishes them.

A good notion of imitates for deterministic algebras is observable equivalence. We say that the algebra-environment pair (C, η_C) *is observably equivalent to* (A, η_A) *with respect to a set of observations OBS* if and only if for all observations $P \in OBS$ with free identifiers from some subset of X, $P(C, \eta_C) = P(A, \eta_A)$.

However, observable equivalence is too strong for nondeterministic algebras. For nondeterministic algebras we want a definition of "imitates" that allows fewer possible results, since in our view a specification only constrains behavior and does not completely determine the exact set of possible results of a nondeterministic program. For example, if we specify that a procedure **g** returns an even number, then an implementation of **g** can have as its set of possible results any nonempty subset of even numbers.

Definition 4.2.1 (imitates). Let OBS be a set of observations, C and A be Σ-algebras, and X be a set of typed identifiers. Let $\eta_C \in ENV(X, C)$ and $\eta_A \in ENV(X, A)$ be environments. Then the pair (C, η_C) *imitates* (A, η_A) *with respect to OBS* if and only if for all observations $P \in OBS$ with free identifiers from some subset of X, $P(C, \eta_C) \subseteq P(A, \eta_A)$.

For example, consider MP-algebras C and A and environments $\eta_C \in ENV(\{\mathbf{x} :$

Mob}, C) and $\eta_A \in ENV(\{\mathtt{x} : \mathtt{Mob}\}, A)$ such that

$$\mathcal{M}[\![\mathtt{next(x)}]\!](C, \eta_C) = \{1,2\} \tag{4.1}$$

$$\mathcal{M}[\![\mathtt{next(x)}]\!](A, \eta_A) = \{1,2,3,4\}. \tag{4.2}$$

Then (C, η_C) imitates (A, η_A) with respect to $\{\mathtt{next(x)}\}$.

When the set of observations is fixed, we simply say that (C, η_C) imitates (A, η_A).

In general, the imitates relation with respect to a fixed set of observations is not symmetric. However, we do have the following easy result.

Lemma 4.2.2. Let *OBS* be a set of observations. Then the imitates relation with respect to *OBS* is reflexive and transitive. ∎

On the other hand, for deterministic algebras, the imitates relation is the same as observable equivalence. For example, consider the IPT-algebra A presented in Figure 2.5 on page 38. Let $X = \{\mathtt{x} : \mathtt{IntPair}\}$. We define two environments $\eta_1, \eta_2 \in ENV(X, A)$ such that $\eta_1(\mathtt{x}) \overset{\text{def}}{=} \langle 1,2,3\rangle$ and $\eta_2(\mathtt{x}) \overset{\text{def}}{=} \langle 1,2\rangle$. We claim that these environments are observably equivalent with respect to the set of observations described by the following set of NOAL programs:

$$\{\mathtt{first(x)},\ \mathtt{second(x)}\}.$$

This claim is verified by considering the following sets of possible results:

$$\mathcal{M}[\![\mathtt{first(x)}]\!](A, \eta_1) = \{1\} \tag{4.3}$$

$$\mathcal{M}[\![\mathtt{second(x)}]\!](A, \eta_1) = \{2\} \tag{4.4}$$

$$\mathcal{M}[\![\mathtt{first(x)}]\!](A, \eta_2) = \{1\} \tag{4.5}$$

$$\mathcal{M}[\![\mathtt{second(x)}]\!](A, \eta_2) = \{2\}. \tag{4.6}$$

Whenever one algebra-environment pair does *not* imitate another with respect to a set of observations, then there is some observation in that set that shows a difference. However, if we think of running the program that defines an observation in a real implementation, we may have to wait forever to "see" the difference, because programs can fail to halt and because we do not make assumptions about how the possible results of nondeterministic operations are chosen. For example, consider MP-algebras C and A and environments $\eta_C \in ENV(\{\mathtt{x} : \mathtt{Mob}\}, C)$ and $\eta_A \in ENV(\{\mathtt{x} : \mathtt{Mob}\}, A)$ such that

$$\mathcal{M}[\![\mathtt{next(x)}]\!](C, \eta_C) = \{1, 2\} \tag{4.7}$$

$$\mathcal{M}[\![\mathtt{next(x)}]\!](A, \eta_A) = \{1\}. \tag{4.8}$$

Then (C, η_C) does not imitate (A, η_A) with respect to $\{\mathtt{next(x)}\}$, but in a real implementation there is no guarantee that the result "2" will be produced in η_C at any time.

The following facts about the how the imitates relation depends on sets of observations will be useful when comparing different programming languages. They are similar to facts about observable equivalence studied by others [ST85, Facts 2–3].

Lemma 4.2.3. Let OBS and OBS' be sets of observations. If $OBS \supseteq OBS'$ and (C, η_C) imitates (A, η_A) with respect to OBS, then (C, η_C) imitates (A, η_A) with respect to OBS'.
∎

That is, the imitates relation with respect to a larger set of observations is a subset of the imitates relation with respect to smaller sets of observations. As an extreme example, the imitates relation with respect to the empty set of observations relates all algebra-environment pairs. In general, adding observations may allow one to observe more differences.

The following says that the imitates relation with respect to a set of observations OBS is the intersection of the imitates relations with respect to all subsets of OBS.

Lemma 4.2.4. Let $OBS = \bigcup_{i \in I} OBS_i$ be a set of observations. If for each $i \in I$, (C, η_C) imitates (A, η_A) with respect to OBS_i, then (C, η_C) imitates (A, η_A) with respect to OBS.
∎

4.2.2 Models of Assertions

We now have all the tools necessary to define when an algebra-environment pair models an assertion with respect to a set of observations.

Definition 4.2.5 (models). Let $SPEC$ be a specification. Let OBS be a set of observations. Let P be a $SPEC$-assertion whose set of free identifiers is X. Let C be a $SPEC$-algebra. Let Y be a set of typed identifiers such that $X \subseteq Y$. Let $\eta_C \in ENV(Y, C)$ be an environment. Then (C, η_C) *models P with respect to OBS*, written $(C, \eta_C) \overset{OBS}{\models} P$, if and only if there is some *SPEC*-algebra A and some nominal environment $\eta_A \in ENV(Y, A)$ such that (C, η_C) imitates (A, η_A) with respect to *OBS* and $\overline{\eta_A}[\![P]\!] = \mathit{true}$.

Notice that the above definition works for all environments, not just those that obey a subtype relation.

We say that (C, η_C) *models P* and write $(C, \eta_C) \models P$ when (C, η_C) models the *SPEC*-assertion P with respect to the set of all type-safe NOAL programs over the nominal signature map and presumed subtype relation of *SPEC*.

As an example of our definition of "models" consider the MP-assertion "b = (2 ∈ m)," where `b` has nominal type `Bool` and `m` has nominal type `Mob`. Let B be an MP-algebra. Let $\eta \in ENV(\{\texttt{b},\texttt{m}\}, B)$ be such that $\eta(\texttt{b}) = \mathit{true}$ and $\eta(\texttt{m})$ is the only possible result of $\mathcal{M}[\![\texttt{ins(make(PSchd,false),2)}]\!](B, \emptyset)$. Then we have

$$(B, \eta) \models \mathrm{b} = (2 \in \mathrm{m}). \tag{4.9}$$

This follows because there is some MP-algebra A and some nominal environment $\eta_A \in ENV(\{\texttt{b},\texttt{m}\}, A)$ such that (B, η) imitates (A, η_A) with respect to type-safe NOAL programs. One type-safe NOAL program is `waiting?(m,2)`, which can have only one possible result, since `waiting?` is specified to be deterministic (in both `Mob` and `PSchd`). Since $\mathcal{M}[\![\texttt{waiting?(m,2)}]\!](B, \eta) = \{\mathit{true}\}$, it follows that $\mathcal{M}[\![\texttt{waiting?(m,2)}]\!](A, \eta_A) = \{\mathit{true}\}$; therefore by the specification of `Mob`, $\overline{\eta_A}[\![(2 \in \mathrm{m})]\!] = \mathit{true}$. Similarly, one can show that $\eta_A(\texttt{b}) = \mathit{true}$. It follows that

$$\overline{\eta_A}[\![\mathrm{b} = (2 \in \mathrm{m})]\!] = \mathit{true}. \tag{4.10}$$

4.3 Semantics of Polymorphic Function Specifications

Now that we have a definition of when an algebra-environment pair models an assertion, it is straightforward to define satisfaction for polymorphic function specifications. To summarize, if the arguments model the precondition, then the function must halt and return a result that models the postcondition. This is a "total-correctness approach" to function specifications.

For concreteness, in this section we define when a NOAL function satisfies a function specification with respect to the set of all type-safe NOAL programs over the referenced specification. The generalization to other applicative programming languages is easy; one uses the definition of "models" with respect to the set of programs for that language. One can even define satisfaction with respect to a subset of a programming language by using that subset as the set of observations.

Although NOAL uses lazy evaluation, we wish to avoid the complications of specifying non-strict functions. Therefore, to define satisfaction we need only be concerned with environments that are proper in the sense that no identifier denotes $\bot$.

Definition 4.3.1 (proper environment). An environment η is *proper* if and only if for every identifier $\mathbf{x}$ in its domain, $\eta(\mathbf{x}) \neq \bot$ (i.e., $\eta(\mathbf{x})$ is proper).

We now define the semantics of NOAL function specifications.

Definition 4.3.2 (satisfies for NOAL functions). Consider the following function specification:

> **fun** f($\vec{\mathtt{x}} : \vec{\mathtt{S}}$) **returns**($\mathtt{v} : \mathtt{T}$)
> **uses** *SPEC*
> **requires** R
> **effect** Q.

Let the presumed subtype relation of *SPEC* be $\leq$. Let $X = \{\mathbf{x}_1 : \mathtt{S}_1, \ldots, \mathbf{x}_n : \mathtt{S}_n\}$. A recursively-defined NOAL function named $\mathtt{f}$ of nominal signature $\vec{\mathtt{S}} \rightarrow \mathtt{T}$ *satisfies* the above specification if and only if for all *SPEC*-algebras C, for all environments $\eta_C \in ENV(X, C)$ such that η_C is proper and obeys $\leq$, the following condition holds. If

$$(C, \eta_C) \models R, \tag{4.11}$$

then for all possible results $q \in \mathcal{M}[\![\mathtt{f}(\vec{\mathtt{x}})]\!](C, \eta_C)$,

$$q \neq \bot \tag{4.12}$$

$$q \in C_{\mathtt{U}} \Rightarrow \mathtt{U} \leq \mathtt{T} \tag{4.13}$$

$$(C, \eta_C[q/\mathtt{v}]) \models Q. \tag{4.14}$$

In the above definition, notice that the possible results must have some type that is a presumed subtype of the nominal result type. This ensures that when a possible result is bound to an identifier of the nominal result type, that binding obeys the presumed subtype relation of the referenced specification. We do not require that the implementation be type-safe, although by using a type-safe implementation one can ensure that the type constraint is met automatically.

Satisfaction for NOAL programs is analogous. However, since a NOAL program does not have formal arguments, we also require that the free identifiers of the program be the same as the formal arguments of the specification it satisfies.

As an example of satisfaction, the NOAL program

```
waiting?(m,2),
```

where `m` has nominal type `Mob`, satisfies the specification `is2waiting` given above. To see this, let C be a MP-algebra, and let $\eta_C \in ENV(\{\mathtt{m} : \mathtt{Mob}\}, C)$ be an environment such that η_C obeys the presumed subtype relation of MP and $\eta_C(\mathtt{m})$. We trivially have that $(C, \eta_C) \models \text{true}$, so the precondition is satisfied. Let $r \in \mathcal{M}[\![\mathtt{waiting?(m,2)}]\!](C, \eta_C)$ be a possible result. Then r is proper, has type `Bool` and is such that

$$(C, \eta_C[r/\mathtt{b}]) \models \mathrm{b} = (2 \in \mathrm{m}). \tag{4.15}$$

This last equation follows from the specification MP, but a formal proof requires the techniques of Chapter 7.

4.4 Discussion

In this section we discuss some limitations of our approach to evaluating assertions, the problems with using our approach to give a semantics to polymorphic operation specifications, and the plausibility of our semantics.

4.4.1 Limitations of Our Techniques for Evaluating Assertions

Our approach to evaluating assertions is similar to the coercer functions found in Bruce and Wegner's work [BW87]. (See Chapter 5 for a detailed comparison.) That is, we "coerce" arguments to the nominal types of the formals and then evaluate the assertions.

Figure 4.4: Specification of the function ins3, which inserts 3 in a scheduler.

fun ins3(m:Mob) **returns**(m3:Mob)
    uses MP
    effect m3 = m $\cup$ {3}

This coercion approach has limitations, although it works for many examples. To illustrate the limitations, consider the function specification ins3, in Figure 4.4. An obvious implementation is the following NOAL function:

```
fun ins3(m:Mob):  Mob = ins(m,3).
```

Notice that if we pass this implementation an object of type PSchd, we get back an object of type PSchd. However, neither the NOAL type system nor the type system of our specification language can express this. One way to do so would be to use a kind of bounded quantification [CW85]. For example we might write something like:

```
fun ins3(m:t ≤ Mob):  t = ins(m,3).
```

This solves the problem with the type of the result, since we can conclude that if ins3 is passed an instance of PSchd, then it returns an instance of PSchd.

However, there is another problem with the coercion approach that is not related to types. Notice that if we pass the denotation of new(PSchd,true) to the above implementation of ins3, we obtain an instance of PSchd whose abstract value is $\langle \mathit{true}, \{3\}\rangle$. However, from the specification of ins3, an implementation can return an instance of PSchd with abstract value $\langle \mathit{false}, \{3\}\rangle$. So our assertion language is not strong enough to state that all other components of an object are unchanged.

Solving these problems is a matter for future work. One possible direction would be to adapt the work of Jategaonkar and Mitchell on ML to specifications [JM88].

4.4.2 Polymorphic Operation Specifications

In this section we discuss the problems that arise in adapting our technique for specifying polymorphic functions to the specification of polymorphic operations in a type specification. (The type specification language described in Chapter 2 can only be used to specify types with operations that are not polymorphic.) Since we do not have clear-cut solutions for these problems, we leave these problems for future work.

The first problem is that our definition of when an algebra-environment pair models an assertion relies on algebras that are known to satisfy a type specification, but we use satisfaction for operation specifications in Chapter 2 to define when an algebra satisfies a type specification. We could perhaps solve this problem by changing our definition of when an algebra-environment pair models an assertion. For example, let Q be a *SPEC*-assertion. Suppose we say that $(C, \eta_C) \overset{OBS}{\models} Q$ if and only if either there is some nominal (C, η'_C) such that (C, η_C) imitates (C, η'_C) with respect to *OBS* and $\overline{\eta'_C}[\![Q]\!] = \mathit{true}$, or there is some algebra A that is known to satisfy *SPEC*, and some nominal η_A such that (C, η_C) imitates (A, η_A) with respect to *OBS* and $\overline{\eta_A}[\![Q]\!] = \mathit{true}$. We could then give an inductive definition of when an algebra satisfies its specification. Algebras that satisfy each operation specification without reference to other algebras can be used as a basis for showing that other algebras satisfy the type specification.

The second problem is that we do not want to make the definition of when an algebra satisfies a type specification dependent on a set of observations, as would be required by the above "solution" to the first problem. It is not clear what set of observations should be used to define satisfaction for operation specifications. For example, should programs that use an operation **g** be considered as observations when defining satisfaction for an implementation of **g**? Furthermore, it is not standard practice to define satisfaction for a type specification with respect to a set of observations. Since we do not want the rest of this dissertation to rest on a nonstandard definition of when an algebra satisfies a specification, we leave these problems for future work.

4.4.3 Plausibility of Our Semantics for Specifications

Why are the above definitions the "right" notions of satisfies and models?

One test is that our definitions specialize to the standard ones when the presumed subtype relation is the identity relation on types (=). The standard definition of when a nominal algebra-environment pair (A, η) models an assertion Q is that $\overline{\eta}[\![Q]\!]$ must be *true*. It is trivial that if a *SPEC*-algebra and a nominal environment are such that the extended environment maps a *SPEC*-assertion to *true*, then the algebra and environment models that assertion, with respect to all sets of observations. However, the converse is not true for all sets of observations. As an extreme example, every satisfiable assertion is modeled by every algebra-environment pair with respect to the empty set of observations. However, the converse does hold if the assertion is observable in the following sense.

To define observable assertions we first define the deterministic observations that characterize their behavior in nominal environments.

Definition 4.4.1 (characteristic observation). Let *OBS* be a set of observations. Let X be a set of typed identifiers. Let Q be a *SPEC*-assertion with free identifiers from X. Then $c_Q \in OBS$ is a *characteristic observation for* Q if and only if c_Q is deterministic, the free identifiers of c_Q are also from X, and for all *SPEC*-algebras A and for all nominal environments $\eta \in ENV(X, A)$,

$$(c_Q(A, \eta) = \{b\}) \;\Leftrightarrow\; (\overline{\eta}[\![Q]\!] = b). \tag{4.16}$$

In an environment that is not proper, the right hand side of the above equation may be $\perp$. When this happens the characteristic observation must have $\perp$ as its only possible result.

It only makes sense to specify the behavior of a characteristic observation on nominal environments, because only nominal environments can be used to evaluate assertions directly. However, since a characteristic observation must be a member of *OBS*, it should be thought of as the denotation of a program written using generic invocation. So we think of a characteristic observation for Q as telling us the meaning of Q, not only for nominal environments but for all environments.

Observable assertions have characteristic observations with respect to some set of observations.

Definition 4.4.2 (observable assertion). Let *OBS* be a set of observations. Let X be a set of typed identifiers. A *SPEC*-assertion Q is *observable with respect to OBS* if and only if there is some $c_Q \in OBS$ that is a characteristic observation for Q.

For convenience, we call a *SPEC*-assertion *observable* if it is observable with respect to the set of all type-safe NOAL programs over the nominal signature map and presumed subtype relation of *SPEC*. A characteristic observation for an observable assertion is thus the denotation of a type-safe NOAL program.

For example, consider the specification MP (`Mob` and `PSchd`) and the MP-assertion "$2 \in$ m" where `m` has nominal type `Mob`. This assertion is observable, since the type-safe NOAL program `waiting?(m,2)` describes a characteristic observation for the assertion "$2 \in$ m."

However, it is easy to write assertions that are not observable. For example, if one specifies a type with a trait function that can give different results for observably equivalent objects, then one can use that trait function to write assertions that are not observable. Furthermore, if one specifies a type whose carrier set can have observably equivalent objects with different representations, then the "=" operator of our specification language can be used to write assertions that are not observable.

Often one uses non-observable assertions because they may give a more concise way of stating the observable properties of an object than an observable assertion. However, since we are ultimately interested in observable properties of specifications, we will often make the simplifying assumption that all assertions are observable.

Our definition of "models" encompasses the standard one for observable assertions. The following lemma states this precisely. It says that a *SPEC*-algebra and a nominal environment model an observable *SPEC*-assertion with respect to a set of observations according to our definition if and only if the pair models the assertion according to the standard definition.

Lemma 4.4.3. Let *SPEC* be a specification. Let *OBS* be a set of observations. Let Q be a *SPEC*-assertion whose free identifiers are a set X.

Suppose Q is observable with respect to *OBS*. Then for all *SPEC*-algebras B and for all nominal environments $\eta_B \in ENV(X, B)$, $(B, \eta_B) \overset{OBS}{\models} Q$ if and only if $\overline{\eta_B}[\![Q]\!] = true$.

Proof: Suppose $\overline{\eta_B}[\![Q]\!] = true$. Then since (B, η_B) imitates itself with respect to *OBS*, by definition $(B, \eta_B) \overset{OBS}{\models} Q$.

Conversely suppose that $(B, \eta_B) \overset{OBS}{\models} Q$. By definition there is some *SPEC*-algebra A and some nominal environment $\eta_A \in ENV(X, A)$ such that (B, η_B) imitates (A, η_A) with respect to *OBS* and $\overline{\eta_A}[\![Q]\!] = true$. Let $c_Q \in OBS$ be a characteristic observation for Q. Since (B, η_B) imitates (A, η_A) with respect to OBS, and the observation c_Q is deterministic, it follows that

$$c_Q(B, \eta_B) = c_Q(A, \eta_A). \tag{4.17}$$

Since $\overline{\eta_A}[\![Q]\!] = true$

$$c_Q(A, \eta_A) = \{true\}. \tag{4.18}$$

Therefore $c_Q(B, \eta_B) = \{true\}$, and thus

$$\overline{\eta_B}[\![Q]\!] = true, \tag{4.19}$$

since c_Q is a characteristic observation for Q. ∎

We can also use characteristic observations to test our definition of "models" for non-nominal environments. The following theorem asserts that our definition of "models" is right in the sense that whenever an algebra-environment pair models an observable assertion with respect to a set of observations, then each characteristic observation for that assertion has *true* as its only possible result.

Theorem 4.4.4. Let $SPEC$ be a specification. Let OBS be a set of observations. Let Q be a $SPEC$-assertion whose free identifiers are a set X. Let B be a $SPEC$-algebra and let $\eta_B \in ENV(X, B)$ be an environment.

If Q is observable with respect to OBS and $(B, \eta_B) \overset{OBS}{\models} Q$, then for each characteristic observation $c_Q \in OBS$ for Q, $c_Q(B, \eta_B) = \{true\}$.

Proof: Suppose that Q is observable with respect to OBS and $(B, \eta_B) \overset{OBS}{\models} Q$. By definition there is some $SPEC$-algebra A and some nominal environment $\eta_A \in ENV(X, A)$ such that (B, η_B) imitates (A, η_A) with respect to OBS and $\overline{\eta_A}[\![Q]\!] = true$. Let $c_Q \in OBS$ be a characteristic observation for Q. Since (B, η_B) imitates (A, η_A) with respect to OBS, and c_Q is deterministic, it follows that

$$c_Q(B, \eta_B) = c_Q(A, \eta_A). \tag{4.20}$$

Since $\overline{\eta_A}[\![Q]\!] = true$ and c_Q is a characteristic observation for Q,

$$c_Q(A, \eta_A) = \{true\}. \tag{4.21}$$

Therefore

$$c_Q(B, \eta_B) = \{true\}. \tag{4.22}$$

∎

The contrapositive of this theorem is important for testing. That is, if an implementation does not pass some test, then we can conclude that the implementation is incorrect. For example, consider a program specification **myProg** with a trivial precondition and referenced specification *SPEC*. Suppose that the postcondition of **myProg** implies some observable assertion Q. (For example, the postcondition might be Q.) Let c_Q be a characteristic observation for Q, let C be a *SPEC*-algebra, and let η_C be an environment that obeys the presumed subtype relation of *SPEC*. Suppose that we are testing a NOAL program P and have found a possible result $q \in \mathcal{M}[\![P]\!](C, \eta_C)$ such that $c_Q(C, \eta_C[q/\mathrm{v}]) = \{false\}$, where v is the formal result of the specification. By the contrapositive of the above theorem, P does not satisfy the specification **myProg**.

The converse of the above theorem is also important for testing, since we want to know when an implementation passes a test case that the implementation satisfies the specification on that test case. Taking the above example again, if for some possible result q, $c_Q(C, \eta_C[q/\mathrm{v}]) = \{true\}$, then we want to know that $(C, \eta_C[q/\mathrm{v}]) \models Q$. Although this is not true for all environments, in the next chapter we define subtype relations so that it is true whenever an environment obeys a subtype relation.

Another test for our definition of models for assertions is whether one's reasoning using the usual laws of propositional calculus is still valid. For example, the law of the excluded middle would say that either $(A, \eta) \overset{OBS}{\models} Q$ or $(A, \eta) \overset{OBS}{\models} \neg Q$, but not both. However, this does not hold in general. That is, if (A, η) does not imitate some nominal algebra-environment pair with respect to *OBS*, then neither $(A, \eta) \overset{OBS}{\models} Q$ nor $(A, \eta) \overset{OBS}{\models} \neg Q$. However, if η obeys a subtype relation with respect to *OBS*, this cannot happen, as we show in the next chapter.

Chapter 5

Subtype Relations

In this chapter we give our formal definition of subtype relations. We also discuss issues, examples, and related work on subtyping.

To reason about programs that exploit inclusion polymorphism, we rely on nominal type information. We write polymorphic function specifications as if each actual argument were an instance of the corresponding formal argument's nominal type. We also verify programs by reasoning about expressions with nominal type `T` as if each possible result was an instance of type `T`. For example, to verify that the NOAL function

```
fun sumFirst(p1,p2: IntPair): Int = add(first(p1),first(p2))
```

implements the specification of Figure 1.4 we reason in the body as if `p1` and `p2` denote instances of `IntPair`, even we can pass instances of subtypes of `IntPair` (such as `IntTriple`) as actual arguments to `sumFirst`.

To guarantee the soundness of reasoning based on nominal type information we ensure that each instance of a subtype of some type `T` imitates some instance of type `T` and that the possible results of each expression of some nominal type `S` are instances of some subtype of `S`. This is the idea behind our formal definition of subtype relations. For example, each instance of type `IntTriple` behaves like an instance of `IntPair` with the same first and second components. By defining subtype relations so that whenever $S \leq T$, each instance of type `S` cannot be observed to behave differently tha ı some instance of type `T`, if we verify some observable property by reasoning about instances of type `T`, then the observable behavior of instances of type `S` will not be surprising.

Our formal definition of subtype relations is parameterized by a set of observations and the semantics of a specification. We define subtype relations with respect to arbitrary sets of observations, so that our definition can be used for many different programming languages. We define subtype relations using the semantics of a specification, so that we can treat subtype relations among ***abstract*** types. We can deal with incompletely specified types, because we take as the semantics of a specification a set of algebraic models. (That is, we take a loose view of specifications.) So our definition of subtype relations is also independent of our specification language.

5.1 Definition of Subtype Relations

Our function specifications allow one to create environments that obey a binary relation on types. The definition of subtype relations ensures that each such environment imitates a nominal environment.

Definition 5.1.1 (subtype relation). Let Σ be a signature and let $SPEC$ be a nonempty set of Σ-algebras. Let OBS be a set of observations. Let $\leq$ be a binary relation on type symbols. Then $\leq$ is a *subtype relation on the types of SPEC with respect to OBS* if and only if for all algebras $C \in SPEC$, there is some $A \in SPEC$ such that for all sets of typed identifiers X and for all environments $\eta_C \in ENV(X, C)$, if η_C obeys $\leq$, then there is some nominal environment $\eta_A \in ENV(X, A)$, such that (C, η_C) imitates (A, η_A) with respect to OBS.

A trivial example of a subtype relation on the types of a specification with respect to a set of observations is the empty relation. Only slightly less trivial is the identity relation on types; the identity relation is always a subtype relation because the imitates relation is reflexive.

Consider the specification IPT presented in Figure 2.2. Let $\leq$ be the smallest reflexive relation on the types of IPT such that **IntTriple** $\leq$ **IntPair**. Then $\leq$ is a subtype relation with respect to the following set of NOAL programs, where **x** : **IntPair**:

$$\{\mathbf{first(x)},\ \mathbf{second(x)}\}.$$

To see this, let C be an IPT-algebra, and let $\eta_C \in ENV(Y, C)$ be an environment that obeys $\leq$. Let $\eta_2 \in ENV(Y, C)$ be defined such that if $\eta_C(\mathbf{x})$ is a proper instance of `IntTriple`, then $\eta_2(\mathbf{x})$ is an instance of `IntPair` such that $C_{\#.\mathrm{first}}(\eta_2(\mathbf{x})) = C_{\#.\mathrm{first}}(\eta_C(\mathbf{x}))$ and $C_{\#.\mathrm{second}}(\eta_2(\mathbf{x})) = C_{\#.\mathrm{second}}(\eta_C(\mathbf{x}))$. For all other identifiers, $\mathbf{y} \in Y$, let $\eta_2(\mathbf{y}) = \eta_C(\mathbf{y})$.

Then (C, η_C) imitates (C, η_2) with respect to the above set of NOAL programs, as is easily checked.

On the other hand, a relation $\leq$ such that **IntPair** $\leq$ **IntTriple** is not a subtype relation with respect to the set of type-safe NOAL programs over the nominal signature map of IPT and $\leq$. To see this, consider the program

```
second(x) & (third(x) & empty(IntStream)),
```

where `x` : `IntTriple`. When this program is applied to an IPT-algebra and a nominal environment that maps `x` to a proper value of type `IntTriple`, the only possible results are streams consisting of two integers (e.g., $\langle 2, 3\rangle$ is the only possible result when the result of `make(IntTriple,1,2,3)` is bound to `x`). However, if this program is applied to an IPT-algebra and an environment where the result of `make(IntPair,1,2)` is bound to `x`, then the partial stream $\langle 2, \perp\rangle$ will result, since applying the instance operation `third` to an instance of `IntPair` results in $\perp$. Therefore, such an algebra-environment pair does not imitate a nominal algebra-environment pair, and hence this $\leq$ is not a subtype relation.

Notice how the set of observations makes a difference in the above example, because with respect to the set of programs

{`first(x)`, `second(x)`}

where `x` has type `IntTriple`, `IntPair` *is* a subtype of `IntTriple`.

In the rest of this section we show how our definition handles nondeterministic and incompletely specified types.

5.1.1 Subtypes can be More Deterministic

In this subsection we present an example that shows how a subtype, `PSchd`, can be more deterministic than its supertype, `Mob`. Therefore, by specifying a supertype with nondeterministic operations, one leaves open implementation decisions that a subtype can make.

The specification of the type `Mob` is given in Figure 2.6 on page 39. The elements of the carrier set of `Mob` can be thought of as finite sets of integers "waiting" to be scheduled.

The ins operation produces a larger Mob object from an existing one by adding its integer argument to the new Mob. The waiting? operation tests for membership in a Mob. The empty? operation tests whether a Mob is empty. The next operation, when applied to a nonempty Mob, is allowed to return any integer waiting in the Mob. Furthermore, the result of applying next to an empty Mob is undefined. If B is a maximally nondeterministic Mob-algebra whose carrier set for the type Mob consists of finite sets of integers, and m is a nonempty finite set of integers, then

$$B_{\mathtt{next}_{\mathtt{Mob}-\mathtt{Int}}}(m) \overset{\text{def}}{=} m \tag{5.1}$$

$$B_{\mathtt{next}_{\mathtt{Mob}-\mathtt{Int}}}(\{\}) \overset{\text{def}}{=} B_{\mathtt{Int}} = \{\bot, 0, 1, -1, \ldots\}. \tag{5.2}$$

The scheduler type PSchd is specified in Figure 4.2 on page 63. It is similar to the type Mob, except that its next operation returns either the least or the greatest integer waiting to be scheduled, with the priority determined by the boolean that is fixed when the object is created. The leastFirst operation returns the priority of a PSchd instance. If B is a PSchd-algebra whose carrier set consists of pairs of booleans and sets of integers, m is a nonempty finite set of integers, and b is either *true* or *false*, then we have

$$B_{\mathtt{next}_{\mathtt{PSchd}-\mathtt{Int}}}(\langle b, m\rangle) \overset{\text{def}}{=} \begin{cases} \min(m) & \text{if } b = \mathit{true}, \\ \max(m) & \text{otherwise.} \end{cases} \tag{5.3}$$

$$B_{\mathtt{next}_{\mathtt{PSchd}-\mathtt{Int}}}(\{\}) \overset{\text{def}}{=} B_{\mathtt{Int}}. \tag{5.4}$$

Let MP be the specification that combines both Mob and PSchd. Let $\leq$ be the smallest reflexive relation on the types of MP such that PSchd $\leq$ Mob. Let *OBS* be the following set of NOAL programs, where **x** : Mob and **i** : **Int**

$$\{\mathtt{waiting?(x, i)},\ \mathtt{empty?(x)},\ \mathtt{next(x)}\}.$$

Let C be an MP-algebra. Let A be an algebra that is the same as C, except that its $\mathtt{next}_{\mathtt{Mob}\to\mathtt{Int}}$ operation exhibits all the nondeterminism allowed by its specification. Then A is an MP-algebra. Let $\eta_C \in ENV(X, C)$ be an environment. Then we can form an environment $\eta_A \in ENV(X, A)$ such that (C, η_C) imitates (A, η_A) with respect to *OBS* as follows. For each $\mathtt{y} \in X$ such that y has nominal type Mob and $\eta_C(\mathtt{y})$ is a proper instance of PSchd, we let $\eta_A(\mathtt{y})$ be an instance of Mob with the same elements; that is, for all $j \in C_{\mathtt{Int}}$, $C_{\#\in\#}(\eta_C(\mathtt{y}), j)$ is true if and only if $A_{\#\in\#}(\eta_A(\mathtt{y}), j)$ holds. For all other

Figure 5.1: Specification of the deterministic scheduler type `Crowd`.

```
Crowd **immutable type**
    **class ops** [new] **instance ops** [ins, waiting?, next]
    **based on sort** C **from** Set **with** [Int **for** T]

    **op** new(c:CrowdClass) **returns**(m:Crowd)
        **effect** m = {}

    **op** ins(c:Crowd, i:Int) **returns**(m:Crowd)
        **effect** m = c ∪ {i}

    **op** waiting?(c:Crowd, i:Int) **returns**(b:Bool)
        **effect** b = i ∈ c

    **op** empty?(c:Crowd) **returns**(b:Bool)
        **effect** b = (c = {})

    **op** next(c:Crowd) **returns**(i:Int)
        **requires** c ≠ {}
        **effect** i ∈ c
```

identifiers $y' \in X$, let $\eta_A(y') = \eta_C(y')$. (This makes sense, because the carrier sets of A and C are the same.) So $\leq$ is a subtype relation on the types of MP with respect to *OBS*.

5.1.2 Incompletely Specified Supertypes

In this subsection we show how our definition of subtype relations handles incompletely specified types. This example shows why our definition is based on the imitates relation between algebra-environment pairs. The example also shows why we chose the particular order of the quantifiers in the definition of subtype relations.

The specification of the type `Mob` is *incomplete*, since some algebras that satisfy that specification are not observably equivalent.

Although the type `Mob` has maximally nondeterministic mo ls that in some sense capture all the behavior of the specification, there are specifications for which no such model exists. One such type is the deterministic scheduler type `Crowd` specified in Figure 5.1.

Figure 5.2: Subtype relationships among the scheduler types.

$$\begin{tikzcd}
 & \text{Mob} & \\
\text{Crowd} \arrow[ur] & & \text{PSchd} \arrow[ul]
\end{tikzcd}$$

The type `Crowd` is similar to the type `Mob`, except that whenever the instance operation `next` is defined, it is required to be deterministic. For example, one `Crowd`-algebra is B^{min}, whose $\mathtt{next}_{\mathtt{Crowd}\rightarrow\mathtt{Int}}$ operation returns the minimum of its argument set.

Let MCP be the specification that combines the specifications of `Mob`, `Crowd`, and `PSchd`. Let $\leq$ be the smallest reflexive relation $\leq$ on the types of MCP such that `Crowd` $\leq$ `Mob` and `PSchd` $\leq$ `Mob`. This relation is depicted in Figure 5.2.

To show that the relation $\leq$ is a subtype relation, when we are given an environment *that obeys* $\leq$ *we find a nominal* environment that the given environment imitates by using a different algebra. For example, let *OBS* be the following set of NOAL programs, where `x` : `Mob` and `i` : `Int`

$$\{\mathtt{waiting?(x,i)},\ \mathtt{empty?(x)},\ \mathtt{next(x)}\}.$$

As in the previous section, if C is an MCP-algebra, $\eta_C \in \mathit{ENV}(X,C)$, η_C obeys $\leq$, and A is an MCP-algebra with the same carrier sets and trait functions as C but with a $\mathtt{next}_{\mathtt{Mob}\rightarrow\mathtt{Int}}$ operation that is as nondeterministic as its specification allows, then we can form an environment $\eta_A \in \mathit{ENV}(X,A)$ such that (C,η_C) imitates (A,η_A) with respect to *OBS*. So $\leq$ is a subtype relation on the types of MCP with respect to *OBS*. We must be able to pick a different an algebra other than C, because the operations of C may not be nondeterministic enough for us to find a nominal environment that a given environment that obeys $\leq$ imitates. Our definition of subtype relations forces us to pick a single algebra A for each C, so that we can reason about an entire program using the same algebra A as a model, without changing algebras for each expression.

Let us consider why we do not want a relation on types such that `PSchd` $\leq$ `Crowd` to be a subtype relation with respect to type-safe NOAL programs over the nominal signature map of MCP and $\leq$. Let η be an environment defined so that the result of `new(PSchd,true)` is bound to `x1` and the result of `new(PSchd,false)` is bound to `x2`, where `x1` and `x2` are both of nominal type `Crowd`. Consider the following program:

```
fun bool2int(b: Bool): Int = if b then 1 else 0 fi;
bool2int(empty?(x1)) & (bool2int(empty?(x2))
& (next(ins(ins(x1,1),2)) & (next(ins(ins(x2,1),2))
& empty(IntStream)))).
```

In the environment η, the only possible result of this program is the stream $\langle 1,1,1,2 \rangle$. In a nominal environment, however, the objects bound to `x1` and `x2` must be instances of type `Crowd`. Clearly, the denotations of `x1` and `x2` in a nominal environment must be empty `Crowd`s if the result of the above program is to be $\langle 1,1,1,2 \rangle$. Using the specification of the trait Set [GH86a, Page 146], whose proper elements are generated by the trait functions $\{\}$ and "insert" and are partitioned by the trait function $\in$, we can conclude that if the denotations of `x1` and `x2` are empty instances of `Crowd`, then they must be the same object and therefore that `ins(ins(x1,1),2)` and `ins(ins(x2,1),2)` must denote the same object. Therefore, since the `next` operation of `Crowd` is deterministic, the last two elements of the stream must be either both 1 or both 2. So presuming that `PSchd` is a subtype of `Crowd` would lead to surprising results. Another way to look at this is that the behavior of these two `PSchd` objects is surprising when they are thought of as `Crowd` objects.

Our definition prevents a relation on types such that `PSchd` $\leq$ `Crowd` from being a subtype relation, because we use imitates for algebra-environment pairs. This allows us to observe more than one object at a time. Although each individual instance of `PSchd` "acts like" some instance of `Crowd`, in general these instances of `Crowd` must be from different algebras. For example, the `PSchd` object created by `new(PSchd,true)` acts like the `Crowd` object created by `new(Crowd)` in B^{min}, since it has the same responses to type-safe NOAL programs that observe it through an identifier of nominal type `Crowd`. Similarly, the `PSchd` object created by `new(PSchd,false)` acts like the `Crowd` object created by `new(Crowd)` in B^{max}. As we saw above, we cannot treat both empty `PSchd` objects as if they were instances of type Crowd, because taken together they do not act like Crowd objects. Therefore, a correct characterization of subtype relations using relationships among objects, such as the one given in the next chapter, requires some "bundling" of the object relationships to properly distinguish between PSchd and Crowd.

5.2 Reasoning about Environments that obey a Subtype Relation

Because our method for evaluating assertions (see Chapter 4) relies on the imitates relation and nominal environments, there are several ways in which standard reasoning could fail. For example, our method of evaluating assertions might be ambiguous, proof by contradiction might be invalid, and we might not be able to conclude that if P and $P \Rightarrow Q$ hold, then Q holds. In this section we show that none of these possibilities can happen if we limit ourselves to reasoning about observable assertions and environments that obey a subtype relation with respect to sufficiently large set of observations. The results of this section complete the plausibility arguments of the last chapter and are used in proving the soundness of our verification techniques.

A fundamental pitfall in our definition of models for assertions is that it might be possible for an assertion to be both true and false, or neither true nor false. In the following lemma we show that observable assertions cannot be both true and false.

Lemma 5.2.1. Let *SPEC* be a specification. Let *OBS* be a set of observations. Let Q be a *SPEC*-assertion whose set of free identifiers is X. Let B be a *SPEC*-algebra and let $\eta_B \in ENV(X, B)$ be an environment.

Suppose that $c_Q \in OBS$ is a characteristic observation for Q and $c_{\neg Q} \in OBS$ is a characteristic observation for $\neg Q$. If (B, η_B) models Q with respect to *OBS* then (B, η_B) does not model $\neg Q$ with respect to *OBS*, and furthermore if (B, η_B) models $\neg Q$ with respect to *OBS* then (B, η_B) does not model Q with respect to *OBS*.

Proof: Suppose for the sake of contradiction that $(B, \eta_B) \overset{OBS}{\models} Q$ and $(B, \eta_B) \overset{OBS}{\models} \neg Q$. By definition, there is some *SPEC*-algebra A and some nominal environment $\eta_A \in ENV(X, A)$ such that (B, η_B) imitates (A, η_A) with respect to *OBS* and

$$\overline{\eta_A}[\![Q]\!] = true. \tag{5.5}$$

By Theorem 4.4.4,

$$c_Q(B,\eta_B) = \{true\} \tag{5.6}$$
$$c_{\neg Q}(B,\eta_B) = \{true\}. \tag{5.7}$$

Since $c_{\neg Q} \in OBS$,

$$c_{\neg Q}(A,\eta_A) = \{true\}. \tag{5.8}$$

Since η_A is nominal, by definition of characteristic observation

$$\overline{\eta_A}[\![\neg Q]\!] = true. \tag{5.9}$$

But this is a contradiction to the definition of evaluation in nominal environments. ∎

The above lemma rules out an observable assertion being both true and false, but it does not guarantee that an assertion is *either* true or false. For example, if an environment does not imitate a nominal environment, then an assertion may be neither true nor false in that environment. However, by restricting our attention to proper environments that obey a subtype relation and sets of observations that can test whether an environment is proper, we can show that the law of the excluded middle holds, as in the following theorem.

Theorem 5.2.2. Let *SPEC* be a specification. Let *OBS* be a set of observations. Let $\leq$ be a binary relation on the types of *SPEC*. Let Q be a *SPEC*-assertion whose set of free identifiers is X. Let B be a *SPEC*-algebra and let $\eta_B \in ENV(X,B)$ be an environment.

Suppose that $\leq$ is a subtype relation on the types of *SPEC* with respect to *OBS*, η_B is proper, η_B obeys $\leq$, $c_Q \in OBS$ is a characteristic observation for Q, $c_{\neg Q} \in OBS$ is a characteristic observation for $\neg Q$, and for all *SPEC*-algebras A and for all nominal environments $\eta_A \in ENV(X,A)$, if (B,η_B) imitates (A,η_A) with respect to *OBS*, then η_A is proper. Then either $(B,\eta_B) \overset{OBS}{\models} Q$ or $(B,\eta_B) \overset{OBS}{\models} \neg Q$, but not both.

Proof: Since η_B obeys $\leq$ and $\leq$ is a subtype relation, there is some *SPEC*-algebra A and some nominal environment $\eta_A \in ENV(X,A)$ such that (B,η_B) imitates (A,η_A) with respect to *OBS*. By hypothesis η_A is proper. So either $\overline{\eta_A}[\![Q]\!] = true$ or $\overline{\eta_A}[\![Q]\!] = false$.

So by definition, either $(B, \eta_B) \overset{OBS}{\models} Q$ or $(B, \eta_B) \overset{OBS}{\models} \neg Q$. By the previous lemma it cannot be both. ∎

So for environments that are proper and obey a subtype relation, we can think of our definition of models for assertions as assigning a single truth value to each observable assertion.

We also want to show that we can reason about environments that obey a subtype relation using the other propositional connectives. To prove the validity of the usual laws that govern the other propositional connectives, we first need to tie the truth value of an assertion in an environment to all the nominal environments it imitates. This is done by the following two lemmas.

The following lemma is a direct consequence of our technique for evaluating assertions and the transitivity of the imitates relation. It says that an algebra-environment pair models each assertion modeled by an algebra-environment pair that it imitates.

Lemma 5.2.3. Let *SPEC* be a specification. Let *OBS* be a set of observations. Let P be a *SPEC*-assertion with free identifiers from X. Let C and A be *SPEC*-algebras. Let Y be a set of typed identifiers such that $X \subseteq Y$. Let $\eta_C \in ENV(Y, C)$ and $\eta_A \in ENV(Y, A)$ be environments.

If (C, η_C) imitates (A, η_A) with respect to *OBS* and $(A, \eta_A) \overset{OBS}{\models} P$, then $(C, \eta_C) \overset{OBS}{\models} P$. ∎

We also need something like a converse to the above lemma. The following lemma is nearly the converse to the above, but the hypothesis is stronger, since the assertion must be observable and the imitated environment must be nominal.

Lemma 5.2.4. Let *SPEC* be a specification. Let *OBS* be a set of observations. Let R be a *SPEC*-assertion whose set of free identifiers is X. Let C and A be *SPEC*-algebras. Let $\eta_C \in ENV(X, C)$ and $\eta_A \in ENV(X, A)$ be environments.

If R is observable with respect to *OBS*, η_A is nominal, (C, η_C) imitates (A, η_A) with respect to *OBS*, and $(C, \eta_C) \overset{OBS}{\models} R$, then $(A, \eta_A) \overset{OBS}{\models} R$.

Proof: Suppose that R is observable with respect to *OBS*, η_A is nominal, (C, η_C) imitates (A, η_A) with respect to *OBS*, and $(C, \eta_C) \overset{OBS}{\models} R$. Since R is observable with respect to OBS, there is some characteristic observation $c_R \in OBS$ for R. By Theorem 4.4.4, $c_R(C, \eta_C) = \{true\}$. Since (C, η_C) imitates (A, η_A) with respect to OBS, and since c_R is deterministic,

$$c_R(C, \eta_C) = c_R(A, \eta_A). \tag{5.10}$$

Since c_R is a characteristic observation and η_A is nominal, $(A, \eta_A) \overset{OBS}{\models} R$. ∎

The following theorem says that the usual way one reasons about implication is valid for environments that obey a subtype relation.

Theorem 5.2.5. Let $SPEC$ be a specification. Let OBS be a set of observations. Let P and Q be $SPEC$-assertions. Let X be a set of typed identifiers that contains the free identifiers of P and Q. Let C be a $SPEC$-algebra and let $\eta_C \in ENV(X, C)$ be an environment.

Suppose that $\leq$ is a subtype relation on the types of $SPEC$ with respect to OBS, that $\neg P$, Q, and $P \Rightarrow Q$ are observable with respect to OBS, and that η_C obeys $\leq$. Then $(C, \eta_C) \overset{OBS}{\models} P \Rightarrow Q$ if and only if either $(C, \eta_C) \overset{OBS}{\models} \neg P$ or $(C, \eta_C) \overset{OBS}{\models} Q$.

Proof: Since $\leq$ is a subtype relation and η_C obeys $\leq$, there is some $SPEC$-algebra A and some nominal environment $\eta_A \in ENV(X, A)$ such that (C, η_C) imitates (A, η_A) with respect to OBS.

Suppose that $(C, \eta_C) \overset{OBS}{\models} P \Rightarrow Q$. Since $P \Rightarrow Q$ is observable with respect to OBS and η_A is nominal, by Lemma 5.2.4,

$$(A, \eta_A) \overset{OBS}{\models} P \Rightarrow Q. \tag{5.11}$$

So by Lemma 4.4.3,

$$\overline{\eta_A}[\![P \Rightarrow Q]\!] = true. \tag{5.12}$$

So by the definition of evaluation in nominal environments, either $\overline{\eta_A}[\![\neg P]\!] = true$ or $\overline{\eta_A}[\![Q]\!] = true$. So by definition, either $(C, \eta_C) \overset{OBS}{\models} \neg P$ or $(C, \eta_C) \overset{OBS}{\models} Q$.

Suppose that $(C, \eta_C) \overset{OBS}{\models} \neg P$. Since $\neg P$ is observable with respect to OBS and η_A is nominal, by Lemma 5.2.4,

$$(A, \eta_A) \overset{OBS}{\models} \neg P. \tag{5.13}$$

So by Lemma 4.4.3,

$$\overline{\eta_A}[\neg P] \ = \ \mathit{true}. \tag{5.14}$$

So by the definition of evaluation in nominal environments, $\overline{\eta_A}[P \Rightarrow Q] = \mathit{true}$, and hence $(C, \eta_C) \overset{OBS}{\models} P \Rightarrow Q$ by definition.

Similarly, if $(C, \eta_C) \overset{OBS}{\models} Q$, then $(C, \eta_C) \overset{OBS}{\models} P \Rightarrow Q$. ∎

Since the set $\{\neg, \Rightarrow\}$ is a complete set of propositional connectives, our usual reasoning about observable assertions is valid in proper environments that obey a subtype relation with respect to a sufficiently large set of observations. For example, standard reasoning about observable assertions in environments that obeys a subtype relation with respect to the set of type-safe NOAL programs is valid, because the NOAL programs of the form `isDef?(x)` ensure that a proper environment can only imitate another proper environment.

Another strange aspect of our definition of models is the way the truth of an assertion in a given environment seems to depend on all the bindings in that environment. This is because the imitates relation depends on all the bindings in an environment and not just those used in some assertion. We show in the following lemma that if we shrink the domain of an environment to excise some identifiers that do not occur free in an assertion, then the truth value of that assertion in the environment is unchanged.

Lemma 5.2.6. Let *OBS* be a set of observations. Let *SPEC* be a specification. Let C be a *SPEC*-algebra. Let Q be a *SPEC*-assertion whose set of free identifiers is X. Let $\eta_C \in ENV(X, C)$ be an environment. Let $\vec{q} \in C_{TYPES}$ be a tuple of objects from the carrier set of C, and let $\vec{\mathbf{z}}$ be a tuple of typed identifiers, none of which is in X.

If $(C, \eta_C[\vec{q}/\vec{\mathbf{z}}]) \overset{OBS}{\models} Q$, then $(C, \eta) \overset{OBS}{\models} Q$.

Proof: Suppose that $(C, \eta_C[\vec{q}/\vec{\mathbf{z}}]) \overset{OBS}{\models} Q$. By definition there is some *SPEC*-algebra A and some nominal environment $\eta_A[\vec{r}/\vec{\mathbf{z}}]$ such that $(C, \eta_C[\vec{q}/\vec{\mathbf{z}}])$ imitates $(A, \eta_A[\vec{r}/\vec{\mathbf{z}}])$ with respect to *OBS* and $\overline{\eta_A[\vec{r}/\vec{\mathbf{z}}]}[Q] = \mathit{true}$. Since the $\mathbf{z}_i$ do not occur free in Q, $\overline{\eta_A}[Q] = \mathit{true}$. Furthermore, (C, η_C) imitates (A, η_A) with respect to *OBS*. So by definition $(C, \eta_C) \models Q$.
∎

If we extend an environment by adding some binding, we may not be able to conclude that an assertion that held in the original environment still holds, because the extended environment may not imitate a nominal environment. But if the extended environment obeys a subtype relation and the assertion is observable, then the truth value of the assertion is unchanged, as we show in the following lemma.

Lemma 5.2.7. Let *OBS* be a set of observations. Let *SPEC* be a specification. Let C be a *SPEC*-algebra. Let P be a *SPEC*-assertion whose set of free identifiers is X. Let $\leq$ be a binary relation on the types of *SPEC*. Let $\eta_X \in ENV(X, C)$ be an environment. Let Y be a set of typed identifiers such that $X \subseteq Y$. Let $\eta_Y \in ENV(Y, C)$ be an environment such that for all $\mathbf{x} \in X$, $\eta_Y(\mathbf{x}) = \eta_X(\mathbf{x})$.

If $(C, \eta_X) \overset{OBS}{\models} P$, η_Y obeys $\leq$, $\leq$ is a subtype relation on the types of *SPEC* with respect to *OBS*, and P is observable with respect to *OBS*, then $(C, \eta_Y) \overset{OBS}{\models} P$.

Proof: Suppose that $(C, \eta_X) \overset{OBS}{\models} P$, η_Y obeys $\leq$, $\leq$ is a subtype relation on the types of *SPEC* with respect to *OBS*, and P is observable with respect to *OBS*. Since $\leq$ is a subtype relation, there is some *SPEC*-algebra A and some nominal environment $\eta'_Y \in ENV(Y, A)$ such that (C, η_Y) imitates (A, η'_Y) with respect to *OBS*. Let η'_X be η'_Y restricted to X. Then (C, η_X) imitates (A, η'_X) with respect to *OBS*. Since the set of free identifiers of P is X,

$$\overline{\eta'_Y}[\![P]\!] = \overline{\eta'_X}[\![P]\!]. \tag{5.15}$$

Since P is observable, $(C, \eta_X) \overset{OBS}{\models} P$, and (A, η'_X) is nominal, by Lemma 5.2.4 $(A, \eta'_X) \overset{OBS}{\models} P$. So by Lemma 4.4.3, $\overline{\eta'_X}[\![P]\!] = \mathit{true}$. So $\overline{\eta'_Y}[\![P]\!] = \mathit{true}$ and thus $(A, \eta'_Y) \overset{OBS}{\models} P$. Since η'_Y is nominal, by Lemma 5.2.3 $(C, \eta_Y) \overset{OBS}{\models} P$. ∎

5.3 Discussion

In this section we first discuss several aspects of subtype relations. We show how subtyping varies with observations. We discuss the meaning of tests on environments that obey a subtype relation and a strategy for testing functions that exploit inclusion polymorphism. Finally we discuss the degrees of freedom one has when designing a collection of types to ensure certain subtype relationships. We have discussed nondeterministic and incomplete specifications above. Continuing this discussion, in the final subsections of this section we discuss how the preconditions of operations (requirements), exceptions, and virtual types affect subtype relations.

5.3.1 How Subtyping Varies with Observations

Like the imitates relation, whether a binary relation on types is a subtype relation varies with the observations one makes. As a trivial example, every binary relation on types is a subtype relation with respect to the empty set of observations. If a feature is added to one's programming language, then some binary relations on types may cease to be subtype relationships with respect to programs written in the new programming language. However, if we remove a feature from a language, by the following lemma, old subtype relations are still subtype relations with respect to the smaller language.

Lemma 5.3.1. Let *SPEC* be a set of Σ-algebras. Let *OBS* and *OBS'* be sets of observations.

If $OBS \supseteq OBS'$, then all subtype relations on the types of *SPEC* with respect to *OBS* are also subtype relations with respect to *OBS'*. ∎

One way to handle an enlarged programming language is suggested by the following lemma. If one knows that $\leq$ is a subtype relation on the types of *SPEC* with respect to OBS_1, then to verify that $\leq$ is a subtype relation with respect to $OBS_1 \cup OBS_2$ one merely has to verify that $\leq$ is a subtype relation with respect to OBS_2.

Lemma 5.3.2. Let *SPEC* be a set of Σ-algebras. Let $OBS = \bigcup_{i \in I} OBS_i$ be a set of observations.

If for each $i \in I$, $\leq$ is a subtype relation on the types of *SPEC* with respect to OBS_i, then $\leq$ is a subtype relation on the types of *SPEC* with respect to *OBS*. ∎

We will have more to say about how certain features of programming languages affect subtype relations in Chapter 9.

5.3.2 Testing Functions that use Subtypes

In this subsection we show that the results of test cases have the usual meaning for functions that use inclusion polymorphism and we offer a strategy for testing such functions.

At the end of Chapter 4 we showed that if an implementation does not pass a test case, then the implementation is incorrect. We will now show that the converse holds: if an implementation passes a test case we can then conclude that the implementation satisfied its specification for that test case. For example, suppose we make the following call to an implementation of `sumFirst`

```
sumfirst(make(IntPair,1,2), make(IntTriple,4,5,6))
```

and the implementation returns 5. This is what we expect from the specification given in Figure 1.4, because the only possible results of

```
first(make(IntPair,1,2))
```

and

```
first(make(IntTriple,4,5,6))
```

are 1 and 4 (respectively). We want to conclude that the implementation satisfies its specification for this test case. This conclusion is valid because we can observe that the result of our test is 5 and that the first components of the two arguments are 1 and 4 (respectively).

We can model a test case as the characteristic observation for a function specification's postcondition. The following lemma says that in an environment that obeys a subtype relation, whenever a characteristic observation for the postcondition returns *true*, the implementation satisfies its specification on that test case.

Lemma 5.3.3. Let $SPEC$ be a specification. Let OBS be a set of observations. Let Q be a $SPEC$-assertion whose set of free identifiers is X. Let B be a $SPEC$-algebra and let $\eta_B \in ENV(X, B)$ be an environment.

If $c_Q \in OBS$ is a characteristic observation for Q, $\leq$ is a subtype relation on the types of $SPEC$ with respect to OBS, η_B obeys $\leq$, and $c_Q(B, \eta_B) = \{true\}$, then $(B, \eta_B) \overset{OBS}{\models} Q$.

Proof: Suppose that $c_Q \in OBS$ is a characteristic observation for Q, $\leq$ is a subtype relation on the types of $SPEC$ with respect to OBS, η_B obeys $\leq$, and $c_Q(B, \eta_B) = \{true\}$. Since $\leq$ is a subtype relation with respect to OBS, there is some $SPEC$-algebra A and some nominal environment $\eta_A \in ENV(X, A)$ such that (B, η_B) imitates (A, η_A) with respect to *OBS*. Since $c_Q \in OBS$,

$$c_Q(B, \eta_B) = \{true\} = c_Q(A, \eta_A). \tag{5.16}$$

Since by definition Q is observable and since η_A is nominal, by Lemma 5.2.4 we have $(A, \eta_A) \overset{OBS}{\models} Q$. Since (B, η_B) imitates (A, η_A) with respect to *OBS*, by Lemma 5.2.3, $(B, \eta_B) \overset{OBS}{\models} Q$. ∎

One implication of the above lemma is that we can use instances of subtypes for test data and interpret the results of such tests normally. This raises the question of whether one *should* use instances of subtypes for test data. The question of what test data to use is important, because during testing one is always faced with limited resources (e.g., time and computers) and so one must make trade-offs among test cases.

A useful test strategy for testing functions that use inclusion polymorphism is to concentrate on nominal test data instead of using instances of subtypes for test data. The reason is that if it can be shown by some test that a function does not meet its specification, then this problem can be uncovered by a test that uses arguments that are instances of the function's nominal argument types. Furthermore, since a subtype might only exhibit some but not all of the behavior of its supertypes, testing with instances of a subtype might fail to uncover certain bugs. For the same reason, when testing a function it is wise to use an implementation of each type that is as nondeterministic as the specification of that type allows.

5.3.3 Subtypes can have Weaker Requirements

Our definition of subtype relations also allows a subtype to be more defined than its supertypes, in the sense that the subtype's **requires** clause may be weaker. For example, consider the type PSchd2, where PSchd2 is exactly like PSchd except that the `next` operation is specified as in Figure 5.3. This specification says that when the argument to `next` is empty, the only possible result is 0. Let P2 be the specification that combines PSchd and PSchd2. Then the smallest reflexive relation $\leq$ on the types of P2 such that PSchd2 $\leq$ PSchd is a subtype relation with respect to the following set of NOAL programs:

$$\{\texttt{next(x)}, \texttt{waiting?(x, i)}, \texttt{leastFirst(x)}\}$$

Figure 5.3: Specification of the type PSchd2, which is more defined than PSchd.

```
PSchd2 immutable type
    class ops [new] instance ops [ins, waiting?, next, leastFirst]
    based on sort C from Pair
        with [Bool for T1, OrderedSet with [Int for T] for T2]

    % new, ins, waiting?, empty?, leastFirst as in Figure 4.2.

    op next(p:PSchd2) returns(i:Int)
        effect ((p.second = {}) ⇒ i=0)
                & ((p.second ≠ {}) ⇒ i ∈ p.second)
                    & (p.first ⇒ lowerBound?(p.second,i))
                    & ((¬p.first) ⇒ upperBound?(p.second,i))
```

where x : PSchd and i : Int. This follows because if C is a P2-algebra and η is an environment such that $\eta(\mathtt{x})$ denotes an empty instance of PSchd2, then $\mathcal{M}[\![\mathtt{next(x)}]\!](A, \eta) = \{0\}$. But if A is a maximally nondeterministic P2-algebra and $\eta_A(\mathtt{x})$ denotes an empty instance of PSchd, then $\mathcal{M}[\![\mathtt{next(x)}]\!](A, \eta) = \{\perp, 0, 1-1, \ldots\}$.

Allowing a subtype to be more defined seems right, since the idea of a **requires** clause is to leave the behavior of an operation undefined when the precondition is not met.

5.3.4 Exceptions and Subtyping

Instead of specifying operations with nontrivial preconditions or arbitrarily defining a result, as was done for PSchd2, another way of dealing with boundary conditions is to specify that an operation should signal an exception.

In this subsection, we again consider operation specifications to be syntactic sugar for operation specifications that return instances of a `OneOf` type (see Chapter 2). For example, when the specification of the type `Mob2` given in Figure 2.8 on page 41 says that the **next** operation signals "empty(nil)" when it is passed an empty instance of `Mob2`, we mean that it returns the result of

```
make_empty(OneOf[normal:Int, empty:Null], nil(Null))
```

when it is passed an empty instance of `Mob2`. Consider the specification MM2 that incorporates both our original `Mob` type and the type `Mob2`. Let $\leq$ be a relation on type symbols such that `Mob2` $\leq$ `Mob`. Then $\leq$ is not a subtype relation on the types of MM2 with respect to type-safe NOAL over the nominal signature map of MM2 and $\leq$. This is because exceptions are distinct from normal results. For example, let **x** have nominal type `Mob`. In an environment where **x** is bound to the result of `new(Mob2)`, the only possible result of the program

```
value[Bool](empty?(x),normal)
& (hasTag?(next(x),normal) & empty(BoolStream))
```

is the stream $\langle \mathit{true}, \mathit{false} \rangle$, since the result of `next(x)` is a `OneOf` with the tag **empty** in this environment. This is not possible in a nominal environment, since the result of `next(x)` can only have tag **normal**. So although we said that the result of **next** when applied to an empty instance of `Mob` is "undefined," there are certain things that an implementation of `Mob` cannot do, such as signalling an exception. The type declarations in a specification are enforced even if a **requires** clause is not satisfied[1]. Therefore, for each instance operation, a subtype cannot have more exceptional results than its supertypes.

The program exhibited in the above example also shows that `Mob` is not a subtype of `Mob2`. However, a subtype can have fewer exceptional results than its supertype if the supertype's specification allows a nondeterministic choice between signalling and returning normal results. For example, consider the type `Mob3` as specified in Figure 5.4. The **next** operation of `Mob3` has a **requires** clause, like `Mob`, but its signature allows the

[1]These type declarations are enforced because our definition of algebras and their operations precludes type violations.

Figure 5.4: Specification of the scheduler type **Mob3**.

Mob3 **immutable type**
    class ops [new] **instance ops** [ins, waiting?, next]
    based on sort C **from** Set **with** [Int **for** T]

    % new, ins, waiting? empty? as in Mob and Mob2.

    ndop next(m:Mob3) **returns**(i:Int) **signals**(empty(Null))
        requires m $\neq$ {}
        effect i $\in$ m

operation to signal, like the `next` operation of `Mob2`. Therefore, the `next` operation of `Mob3` can return any element of the carrier set of `OneOf[normal: Int, empty: Null]` when the requires clause is not satisfied.

Let M3 be a specification that combines the types `Mob2` and `Mob3`. Let $\leq$ be the smallest reflexive relation on the types of M3 such that `Mob2` $\leq$ `Mob3`. Let *OBS* be the following set of NOAL programs, where `x` : `Mob` and `i` : `Int`

$$\{\texttt{waiting?}(\texttt{x},\texttt{i}),\ \texttt{empty?}(\texttt{x}),\ \texttt{next}(\texttt{x})\}.$$

Then $\leq$ is a subtype relation with respect to *OBS*. This is another instance of a subtype being more defined than its supertypes.

We can also show that the type `Mob`, which has fewer exceptional results, is a subtype of `Mob3`. Let M4 be a specification that combines the types `Mob` and `Mob3`.

To understand why `Mob` is a subtype of `Mob3` we need to understand the subtype relationships on the `OneOf` types that we use to model exceptions.

A specification of the type `OneOf[normal:Int, empty:Null]` is given in Figure 5.5, where the type name is abbreviated to `NE` and we have taken the liberty of using special syntax for defining the set of instance operations named `value[`T`]` for types T. The trait used to define the carrier set and trait functions of this type is found in Figure 2.7 on page 40. The specification of `OneOf[normal:Int]` is similar.

There are other ways to specify `OneOf` types so that `OneOf[normal:Int]` is a subtype of `OneOf[normal:Int, empty:Null]`. We could include in our programming language built-in expressions for observing `OneOf` instances [LAB*81, Section 11.6] [CW85] [Car84].

Figure 5.5: Specification of the type NE = OneOf[normal: Int, empty: Null].

```
NE immutable type
    class ops [make_normal, make_empty]
        instance ops [hasTag?, value[T]]
    based on sort NE from OneOf[normal: Int, empty: Null]

    op make_normal(c:NEClass, i: Int) returns(o:NE)
        effect o = make_normal(i)

    op make_empty(c:NEClass, n: Null) returns(o:NE)
        effect o = make_empty(n)

    op hasTag?(o:NE, t: Tag) returns(b:Bool)
        effect b = hasTag?(o, t)

    op value[T](o:NE, t: Tag) returns(r:T)
        requires hasTag?(o, t)
            & ((t = normal) ⇒ Int = T)
            & ((t = empty) ⇒ Null = T)
        effect ((t = normal) ⇒ r = val_normal(o))
            & ((t = n₂) ⇒ r = val_empty(o))
```

We could specify `value_ni` operations for each tag `ni`, and limit the set of observations so that each `value_ni` operation can only be invoked on instances that are known to have the tag `ni` [vWMP*77]. These choices lead to the appropriate subtype relationship, since they all allow an observation intended for an instance of `OneOf[normal:Int, empty:Null]` to be applied to an instance of `OneOf[normal:Int]` without surprises. The alternative we have adopted has the advantage of allowing `OneOf` types to be specified without adding special features to a programming language.

Let $\leq$ be the smallest reflexive relation on the types of M4 such that $\texttt{Mob} \leq \texttt{Mob3}$ and

$$\texttt{OneOf[normal : Int]} \leq \texttt{OneOf[normal : Int, empty : Null]}.$$

Let *OBS* be the following set of NOAL programs, where $\texttt{x} : \texttt{Mob3}$, $\texttt{i} : \texttt{Int}$, $\texttt{o} : \texttt{OneOf[normal : Int, empty : Null]}$, and $\texttt{t} : \texttt{Tag}$,

$$\left\{ \begin{array}{c} \texttt{waiting?(x, i), empty?(x), value[Int](next(x), normal),} \\ \texttt{hasTag?(o, t), value[Int](o, normal)} \end{array} \right\}.$$

Let C be an M4-algebra and let $\eta_C \in ENV(X, C)$ be an environment that obeys $\leq$. Let A be an M4-algebra with the same carrier sets and abstract functions as C but with a maximally nondeterministic $\texttt{next}_{\texttt{Mob3}\rightarrow\texttt{Int}}$ operation. Define the environment $\eta_A \in ENV(X, A)$ as follows. If $\eta_C(\texttt{x})$ is a proper instance of `Mob`, then let $\eta_A(\texttt{x})$ be an instance of `Mob3` in A with the same elements; that is, for all $j \in C_{\texttt{Int}}$, $C_{\#\in\#}(\eta_C(\texttt{x}), j)$ is true if and only if $A_{\#\in\#}(\eta_A(\texttt{x}), j)$ holds. If $\eta_C(\texttt{o})$ is a proper instance of `OneOf[normal:Int]`, then let $\eta_A(\texttt{o})$ be an instance of `OneOf[normal:Int, empty:Null]` in A with the same tag (`normal`) and value. Otherwise for all $\texttt{y} \in Y$, let $\eta_A(\texttt{y}) = \eta_C(\texttt{y})$. It is straightforward to show that (C, η_C) imitates (A, η_A) with respect to *OBS*. Thus $\leq$ is a subtype relation on the types of M4 with respect to *OBS*.

In general a subtype (such as `Mob`) can have fewer exceptions than its supertypes (such as `Mob3`), because a `OneOf` type with fewer tags can be a subtype of a `OneOf` type with the same tags and more (as noted by Cardelli [Car84]).

5.3.5 Virtual Supertypes

In many practical examples of object-oriented design, one specifies types without class operations to be used as supertypes. Since these types do not have class operations they

Figure 5.6: The specification Vehicles, including types Vehicle and Bicycle.

```
Vehicle virtual type
    instance ops [wheels, passengers]
    based on sort C from Vehicle

    op wheels(v:Vehicle) returns(i:Int)
        effect i = wheels(v)

    op passengers(v:Vehicle) returns(i:Int)
        effect i = passengers(v)

Bicycle immutable type
    class ops [new] instance ops [wheels,passengers,maker]
    based on sort String from CharString

    op make(BicycleClass, s:String) returns(b:Bicycle)
        effect b = s

    op wheels(b:Bicycle) returns(i:Int)
        effect i = 2

    op passengers(b:Bicycle) returns(i:Int)
        effect i = 1

    op maker(b:Bicycle) returns(s:String)
        effect s = b
```

cannot be instantiated. We call a type that cannot be instantiated a ***virtual type***, since its implementations often use virtual operations. A ***virtual operation*** has an implementation that uses some primitive (called **virtual** in Simula 67 [DMN70] [BDMN73]) to invoke an operation of a subclass [2]; hence a virtual operation cannot be executed unless the subclass has defined the required operation.

Consider the specification Vehicles, given in Figure 5.6. In this specification, Vehicle is a virtual type and has no class operations. The carrier set for Vehicle is described in the trait Vehicle found in Figure 5.7.

We will now show informally that the smallest reflexive relation $\leq$ on the types of

[2]Some researchers (e.g., [SCW85, Page 42] [Sym84, Page 450]) call a type or class that cannot be instantiated "abstract," but this leads to confusion with the term "abstract data type."

Figure 5.7: The trait Vehicle that describes the abstract values of `Vehicle`.

```
Vehicle trait
    introduces
        wheels: C → Int
        passengers: C → Int
    asserts for all [v: C]
        wheels(v) ≥ 1
        passengers(v) ≥ 1
```

Vehicles such that `Bicycle` $\leq$ `Vehicle` is a subtype relation with respect to the set of observations *OBS* defined by the NOAL programs `wheels(x)` and `passengers(x)`, where `x` has nominal type `Vehicle`. Let C be a Vehicles-algebra and let $\eta_C \in ENV(X, C)$ be an environment that obeys $\leq$. Let A be a Vehicles-algebra with the same carrier set for all types as C, except that we add an element q to the carrier set of `Vehicle` if necessary with the property that $A_{\mathbf{wheels}}(q) = 2$ and $A_{\mathbf{passengers}}(q) = 1$. We define a nominal environment $\eta_A \in ENV(X, A)$ as follows. Suppose $\eta_C(\mathbf{x})$ is a proper element of type `Bicycle`; then let $\eta_A(\mathbf{x})$ be the instance $q \in A_{\mathtt{Vehicle}}$ that by assumption is such that $A_{\mathbf{wheels}}(q) = 2$ and $A_{\mathbf{passengers}}(q) = 1$. Otherwise, for all other $\mathbf{y} \in X$, let $\eta_A(\mathbf{y}) = \eta_C(\mathbf{y})$, which makes sense, since the carrier sets for C are subsets of those for A. Then (C, η_C) imitates (A, η_A) with respect to *OBS*, because if η_C maps the identifier `x` of nominal type `Vehicle` to a proper instance of type `Bicycle`, then

$$\mathcal{M}[\![\mathtt{wheels(x)}]\!](C, \eta_C) = \{2\} \tag{5.17}$$

$$\mathcal{M}[\![\mathtt{passengers(x)}]\!](C, \eta_C) = \{1\} \tag{5.18}$$

$$\mathcal{M}[\![\mathtt{wheels(x)}]\!](A, \eta_A) = \{2\} \tag{5.19}$$

$$\mathcal{M}[\![\mathtt{passengers(x)}]\!](A, \eta_A) = \{1\}. \tag{5.20}$$

So `Bicycle` is a subtype of `Vehicle`.

5.4 Other Definitions of Subtype

In this section we compare our definition of subtype relations with other notions of subtyping.

The major difference is that our definition is based on the semantics of *abstract* data types, which allows us to deal with incompletely specified types and with nondeterministic types. The only other formal treatment of subtype relationships that deals with abstract data types is the work of Bruce and Wegner, which is discussed below. Even informal discussions of subtype relationships have largely ignored incompletely specified types.

Another difference is that our definition of subtype relations explicitly takes into account a set of observations. This is a feature of some work on observational equivalence (e.g., [ST85]), but does not appear in other work on subtyping.

5.4.1 Informal Definitions of Subtype Relationships

Schaffert *et al.* offer the following informal definition of a subtype relationship: "Given a type `S` which is a subtype of a type `T`, then any object of type `S` behaves like a `T` object and may be used wherever a `T` object may be used" [SCB*86, Section 5]. For types that are not incompletely specified, this definition seems to agree with our definition of subtyping. We will make this connection more formally in the next chapters, where we study imitation and simulation relations between objects. However, for incompletely specified types a simple relation on objects does not suffice to prove a subtype relation, as we showed above for the types `PSchd` and `Crowd`. In that example, each `PSchd` object acts like some `Crowd` object, but `PSchd` is not a subtype of `Crowd`.

Snyder [Sny86b, Page 41] offers the following definition of a subtype relationship: "If instances of class x meet the external interface of class y, then x should be a subtype of y." By "external interface" Synder means a behavioral specification; thus his definition of subtype relationships is also a semantic relationship between instances. For example, he says that "behavioral subtyping cannot be deduced without formal semantic specification of behavior." However, he goes on to say that "lacking such specifications, one can deduce subtyping based solely on syntactic external interfaces (i.e., the names of the operations) [Car84]." We strongly disagree with this latter statement, since for valid reasoning based on subtype relationships among abstract types, the semantics of a type must be taken into account. For example, consider a type `IntPairLoop`, which is like `IntPair` except that its second operation goes into an infinite loop when called. `IntPairLoop` is not a subtype of `IntPair`, although the two types have compatible syntactic interfaces. Although Snyder cites Cardelli's paper [Car84] to support his statement, Cardelli's syntactic deductions do not apply to abstract types in general, but only to a limited set of types (as we discuss below).

5.4.2 Cardelli's "Semantics of Multiple Inheritance"

Cardelli's "A Semantics of Multiple Inheritance" [Car84] describes subtype relationships among the built-in types of a small programming language, as well as its type checking and semantics. Unlike our definition of subtype relations, Cardelli's paper does not deal with abstract data types. Since Cardelli only deals with a fixed set of built-in types, he is able to give simple syntactic rules that determine when one of these types is a subtype of another type.

Cardelli makes no claim that his syntactic rules extend to abstract types. He only shows that his syntactic subtyping rules are sound in the sense that they prevent certain errors in programs written his language. Cardelli writes $\mathtt{S} \leq \mathtt{T}$ when $\mathtt{S}$ is syntactically a subtype of $\mathtt{T}$. The semantics of Cardelli's language are described using a domain $\mathbf{V}$. This domain is constructed so that, whenever $\mathtt{S} \leq \mathtt{T}$, then the carrier set of $\mathtt{S}$ is a subset of the carrier set of $\mathtt{T}$ [Car84, Page 62]. This supports inclusion polymorphism without generic invocation, since the instance operations of the built-in types of Cardelli's language work on all subsets of their domains, and hence on all subtypes.

We can regard Cardelli's $\mathbf{V}$ as the only "algebra" in the semantics of a specification of the built-in types of his language, although it is not technically one of our algebras. An expression written in his language defines a (deterministic) observation of $\mathbf{V}$. Because of the way that $\mathbf{V}$ is constructed, all environments whose range is $\mathbf{V}$ that obey the relation $\leq$ are also nominal environments. (That is, if $\eta(\mathtt{x})$ has type $\mathtt{S}$ and $\mathtt{x}$ has nominal type $\mathtt{T}$, then $\mathtt{S} \leq \mathtt{T}$, which means that $\mathbf{V}_{\mathtt{S}} \subseteq \mathbf{V}_{\mathtt{T}}$, and so $\eta(\mathtt{x})$ also has type $\mathtt{T}$.) Since every algebra-environment pair imitates itself with respect to all sets of observations written in his language, Cardelli's $\leq$ is a subtype relation (in our sense) with respect to all sets of observations written in his language. So our definition of subtype relations is compatible with Cardelli's. The differences are that Cardelli considers function types, which are not easily modeled by our algebras, and our algebras can model abstract data types, and nondeterministic types, which Cardelli does not handle.

5.4.3 Bruce and Wegner's Definition of Subtyping

The work that comes closest to our own is that of Bruce and Wegner [BW87]. In this subsection we show that our definition of subtype relations is more general than theirs and point out a failing of their definition.

Our definition is more general than Bruce and Wegner's for two reasons. First, our definition handles nondeterministic and incompletely specified types, whereas Bruce and Wegner only deal with a single deterministic algebra. Second, Bruce and Wegner do not define subtype relations with respect to a set of observations. Bruce and Wegner's definition does not even focus on the observable behavior of an algebra; instead they define subtype relations with the aid of a family of homomorphic functions between the carrier sets of an algebra. Since the existence of a homomorphic function is a strong condition, using Bruce and Wegner's definition of subtype relations would exclude some subtype relations that our definition would allow. As we will illustrate below, Bruce and Wegner's definition fails to show a subtype relationship between two types that are observably equivalent. In Chapter 6 we show that certain homomorphic *relations* can be used to prove subtype relationships. Because homomorphic functions are a special case of homomorphic relations, Bruce and Wegner's definition is not as general as ours.

In the rest of this subsection we show how Bruce and Wegner's definition fails to show a subtype relationship between two types that are observably equivalent. We show this with an example adapted from a paper by Mitchell [Mit86, Page 266]. In this example, the types S1 and S2 represent multi-sets of integers. Bruce and Wegner do not tell us how to model specifications, so we simply consider an algebra A, where the elements of the carrier set of S1 are lists (written $[1,2]$ below) of ordered pairs of the form $\langle element, count\rangle$ and the elements of the carrier set of S2 are just lists of the elements inserted in the order in which they are inserted. Besides the class operations new of types S1 and S2 (which return empty multi-sets of types S1 and S2 respectively), each type supports the instance operations ins and count. To illustrate the representations, suppose $\eta(\mathtt{x})$ denotes the result of new(S2). Then we have,

$$\mathcal{M}[\![\mathtt{ins(ins(x,5),40)}]\!](A,\eta) = \{[5,40]\}.$$

Similarly, if $\eta(\mathtt{y})$ denotes the result of `new(S1)`, then

$$\mathcal{M}[\![\mathtt{ins(ins(x,5),40)}]\!](A,\eta) = \{[\langle 5,1\rangle, \langle 40,1\rangle]\}.$$

Our example exploits the differences in these representations. Let the environment η and objects q and r be such that

$$\eta(\mathtt{q}) = q \neq r = \eta(\mathtt{r}).$$

We then have

$$\begin{aligned}
&\mathcal{M}[\![\mathtt{ins(ins(ins(new(S2),q),r),q)}]\!](A,\eta)\\
&= \{[q,r,q]\}\\
&\neq \{[q,q,r]\}\\
&= \mathcal{M}[\![\mathtt{ins(ins(ins(new(S2),q),q),r)}]\!](A,\eta)
\end{aligned}$$

although if we use `S1` instead of `S2` these are equal:

$$\begin{aligned}
&\mathcal{M}[\![\mathtt{ins(ins(ins(new(S1),q),r),q)}]\!](A,\eta)\\
&= \{[\langle q,2\rangle, \langle r,1\rangle]\}\\
&= \mathcal{M}[\![\mathtt{ins(ins(ins(new(S1),q),q),r)}]\!](A,\eta).
\end{aligned}$$

For `S1` to be a subtype of `S2` according to Bruce and Wegner's definition, there would have to be a homomorphic function $c_{(1,2)}$ from `S1` to `S2`. Because of the structure-preserving nature of this function we have the following equations, for all environments η such that $\eta(\mathtt{q}) = q \neq r = \eta(\mathtt{r})$.

$$\begin{aligned}
c_{(1,2)}(\mathcal{M}[\![\mathtt{new(S1)}]\!](A,\eta)) = \mathcal{M}[\![\mathrm{new(S2)}]\!](A,\eta) &= \{[]\}\\
c_{(1,2)}(\mathcal{M}[\![\mathtt{ins(new(S1),q)}]\!](A,\eta)) = \mathcal{M}[\![\mathtt{ins(new(S2),q)}]\!](A,\eta) &= \{[q]\}\\
c_{(1,2)}(\mathcal{M}[\![\mathtt{ins(ins(new(S1),q),r)}]\!](A,\eta)) &= \{[q,r]\}\\
c_{(1,2)}(\mathcal{M}[\![\mathtt{ins(ins(ins(new(S1),q),r),q)}]\!](A,\eta)) &= \{[q,r,q]\}\\
c_{(1,2)}(\mathcal{M}[\![\mathtt{ins(ins(new(S1),q),q)}]\!](A,\eta)) &= \{[q,q]\}\\
c_{(1,2)}(\mathcal{M}[\![\mathtt{ins(ins(ins(new(S1),q),q),r)}]\!](A,\eta)) &= \{[q,q,r]\}.
\end{aligned}$$

Since in η the only possible result of both `ins(ins(ins(new(S1), q), r), q)` and `ins(ins(ins(new(S1), q), q), r)` is the list of pairs $[\langle q, 2\rangle, \langle r, 1\rangle]$, $c_{(1,2)}$ would have to map $[\langle q, 2\rangle, \langle r, 1\rangle]$ to both $[q, r, q]$ and $[q, q, r]$. Therefore, $c_{(1,2)}$ cannot be a function.

So Bruce and Wegner's definition fails to show a subtype relationship between observably equivalent types.

Chapter 6

Simulation Relations

In this chapter we show how to prove that a binary relation is a subtype relation with respect to type-safe NOAL programs using simulation relationships among objects. We also begin to turn our attention from specification to verification, because simulation plays a central role in our Hoare-style verification technique for NOAL programs.

How can one prove that a binary relation on types is a subtype relation with respect to the set of all type-safe NOAL programs? Since the set of all type-safe NOAL programs is infinite, the obvious answer is to use an induction on the structure of NOAL expressions. The obvious induction hypothesis is that if one environment imitates another, then for each subexpression E, each possible result of E in the first environment "imitates" some possible result of E in the second environm nt. However, it is difficult to make this induction work, because imitation only directly constrains the possible results of whole programs, not expressions.

Since we do not know if the obvious induction hypothesis can be made to work, we use a more convenient hypothesis, which may also be stronger. We define a notion of simulation that is preserved by each NOAL expression. So to prove that a binary relation on types is a subtype relation we only have to show that each environment that obeys the relation simulates some nominal environment.

However, we also want simulation to be a more practical tool than imitation. So simulation relations are defined as relations between objects instead of relations between environments. Furthermore, we define simulation relations so that one can easily check whether a relation among objects is a simulation relation. That is, instead of checking that a simulation relation is preserved by all expressions, one only has to check that it is the identity relation on instances of the visible types (like `Bool` and `Int`) and that it is preserved by generic invocation expressions.

For example, consider the specification MP (which contains the types `Mob` and `PSchd` as specified in Figures 2.6 and 4.2) and the relation $\mathcal{R}$ that is the identity on instances of `Bool` and `Int`, and that relates each instance of `PSchd` to a maximally nondeterministic instance of `Mob` with the same set of waiting integers. This $\mathcal{R}$ relates the `PSchd` instance denoted by

```
ins(ins(new(PSchd,true),1),2)
```

to a maximally nondeterministic instance of `Mob` denoted by

```
ins(ins(new(Mob),1),2).
```

We check that this relationship is preserved by each generic invocation in the following manner.

- For the generic operation `empty?`, the only possible result on the `PSchd` instance is *false*, which is also the only possible result on the `Mob` instance.
- For the generic invocation `next`, the only possible result on the `PSchd` instance is 1, which is related by $\mathcal{R}$ to 1, and 1 is a possible result on the `Mob` instance.
- For all integers `i`, the only possible result of

  ```
  waiting?(ins(ins(new(PSchd,true),1),2),i)
  ```

  is related by $\mathcal{R}$ to the only possible result of

  ```
  waiting?(ins(ins(new(Mob),1),2),i).
  ```

- Furthermore, for all integers `j`, the result of

  ```
  ins(ins(ins(new(PSchd,true),1),2),j)
  ```

  is related by $\mathcal{R}$ to the result of

  ```
  ins(ins(ins(new(Mob),1),2),j),
  ```

  since both have the same set of waiting integers.

In this example we have checked only that $\mathcal{R}$ is preserved by the instance operations of type `Mob`. We did not check the `PSchd` instance operation `leastFirst`, because `leastFirst` cannot be applied to an expression of nominal type `Mob`.

In a program that uses inclusion polymorphism, instances of a subtype can be treated as if they were instances of a supertype. For example, suppose `x` and `y` are identifiers of nominal type `Mob` that denote the `PSchd` instances `new(PSchd,true)` and `new(PSchd,false)` (respectively). We can apply only the instance operations of type `Mob` to these identifiers in a type-safe NOAL program; therefore the values of `x` and `y` cannot be distinguished by a type-safe program. However, if `x` and `y` had type `PSchd`, we could also apply the operation `leastFirst`, which would distinguish them. That is, viewed as instances of `Mob`, these instances simulate each other, but viewed as instances of `PSchd` neither simulates the other. Therefore, in our formal treatment of simulation a simulation relation is a family of relations, one per type, so that we can keep the simulation relationships for each view straight. For example, we say that `new(PSchd,true)` simulates `new(PSchd,false)` at type `Mob`, but this relationship does not hold at type `PSchd`.

To prove that a binary relation $\leq$ is a subtype relation, we need a simulation relation such that whenever `S` $\leq$ `T`, then each instance of type `S` is related to some instance of type `T` at type `T`. Such simulation relations are said to *witness* the relation $\leq$. If we have a simulation relation that witnesses $\leq$, then from every environment that obeys $\leq$ we can build a nominal environment that the first environment imitates. The nominal environment is built by replacing each binding of an instance q of type `S` to an identifier of nominal type `T` where `S` $\leq$ `T` with a binding of an instance r of type `T` such that q simulates r at type `T`.

We also use simulation relations that witness a subtype relation during the verification of NOAL programs, where such simulation relations are used to coerce objects from subtypes to supertypes. For example, consider the following call to the function `is2waiting`, which is specified in Figure 4.1:

```
    is2waiting(new(PSchd,true)).
```

From the specification of `PSchd`, we can conclude that if `x` : `PSchd` denotes the result of `new(PSchd,true)`, then ($2 \in$ `x`.second) = false. The specification of `is2waiting` has a description of the function's effect written using the trait functions of type `Mob` (it says that the result is $2 \in \mathtt{m}$, where $\mathtt{m} : \mathtt{Mob}$ is the formal argument). So to find the effect of the call we note that `new(PSchd,true)` simulates `new(Mob)` at type `Mob` and that if $\mathtt{m} : \mathtt{Mob}$ denotes `new(Mob)`, then $(2 \in \mathtt{m}) = \text{false}$. Details appear in Chapter 7.

In the rest of this chapter we give a formal definition of simulation relations, show that simulation is preserved by NOAL expressions, discuss how simulation relations can be used to prove that a binary relation on types is a subtype relation, and give several examples of such proofs.

6.1 Definition of Simulation Relations

In this section we give a formal definition of simulation relations and compare simulation relations with some related concepts.

We call a family of relations, which has one binary relation between the carrier sets of two algebras per type, a typed family of relations.

Definition 6.1.1 (typed family of relations). Let $\Sigma = (TYPES, V, TFUNS, OPS)$ be a signature. A *typed family of relations*, $\mathcal{R}$, between Σ-algebras C and A is a set of binary relations on the carrier sets of C and A indexed by $TYPES$:

$$\mathcal{R} = \{\mathcal{R}_{\mathtt{T}} \subseteq (C_{TYPES} \times A_{TYPES}) \mid \mathtt{T} \in TYPES\}$$

Each $\mathcal{R}_{\mathtt{T}}$ may relate objects of types other than `T` and may also relate objects having different types. For example, $\mathcal{R}_{\mathtt{Mob}}$ may related two instances of `PSchd` or an instance of `PSchd` to an instance of type `Mob`.

To find whether an environment is related by a typed family of relations to some other environment, we use the nominal type of each identifier to select the appropriate relation, $\mathcal{R}_{\mathtt{T}}$. That is, given environments $\eta_1 \in ENV(X, C)$ and $\eta_2 \in ENV(X, A)$, we write $\eta_1 \, \mathcal{R} \, \eta_2$ when for all types `T` and for all identifiers $\mathtt{x} : \mathtt{T} \in X$, $\eta_1(\mathtt{x}) \, \mathcal{R}_{\mathtt{T}} \, \eta_2(\mathtt{x})$.

For notational convenience, we also use the following abbreviations when dealing with a typed family of relations, $\mathcal{R}$, and the binary relations $\mathcal{R}_{\mathtt{T}}$.

- Given subsets $Q \subseteq C_{TYPES}$ and $R \subseteq A_{TYPES}$, we write $Q \, \mathcal{R}_{\mathtt{T}} \, R$ when for all $q \in Q$, there is some $r \in R$ such that $q \, \mathcal{R}_{\mathtt{T}} \, r$. We often use this notation when comparing sets of possible results and the carrier sets of the various types. For example, $C_{\mathtt{S}}\ \mathcal{R}_{\mathtt{T}}\ A_{\mathtt{T}}$ means that for each instance q of type $\mathtt{S}$ in the algebra C, there is an instance r of type $\mathtt{T}$ in A such that $q\ \mathcal{R}_{\mathtt{T}}\ r$.

- Given a tuple $\vec{\mathtt{T}}$ of types and two tuples of objects the same length $\vec{q}$ and $\vec{r}$, we write $\vec{q}\,\mathcal{R}_{\vec{\mathtt{T}}}\,\vec{r}$, when for each i, $q_i\ \mathcal{R}_{\mathtt{T}_i}\ r_i$.

We also use vector notation for tuples of variables and expressions. We also write $\eta(\vec{\mathbf{x}})$ for the tuple of the values, $\langle \ldots, \eta(\mathbf{x}_i), \ldots \rangle$.

There are four properties that characterize when a typed family of relations is a simulation relation. First, a simulation relation is preserved by each generic invocation; we call such a typed family of relations a "homomorphic relation for *NomSig* and $\leq$," since the set of generic invocations and their nominal types are determined by a nominal signature map *NomSig* and a presumed subtype relation $\leq$. Second, a simulation relation must be the identity on the visible types; we call such relations "V-identical." Third, a simulation relation must preserve the meaning of $\bot$; we call such a relation "bistrict." Finally, the relationships at each subtype must hold at each supertype. The last property has a largely technical motivation, but it embodies the intuition that if one object simulates another at a subtype, then this simulation relationship should hold at each supertype, since no other generic operations will be applicable at the supertype.

We give the formal definition of these terms after presenting our definition of simulation relations.

Definition 6.1.2 (simulation relation). Let Σ be a signature. Let *NomSig* be a nominal signature map. Let $\leq$ be a binary relation on the types of Σ. Let C and A be Σ-algebras. A *simulation relation between C and A for NomSig and* $\leq$ is a homomorphic relation between C and A for *NomSig* and $\leq$ that is V-identical, bistrict and such that for all types $\mathtt{S}$ and $\mathtt{T}$,

$$(\mathtt{S} \leq \mathtt{T}) \Rightarrow (\mathcal{R}_{\mathtt{S}} \subseteq \mathcal{R}_{\mathtt{T}}) \qquad (6.1)$$

The definition of a homomorphic relation formalizes the notion that the relation at each type is preserved by each generic invocation that is applicable to expressions of that type.

Definition 6.1.3 (homomorphic relation). Let $\Sigma = (\mathit{TYPES}, V, \mathit{TFUNS}, \mathit{OPS})$ be a signature. Let *NomSig* be a nominal signature map whose domain is a set *GOP* of generic operation symbols. Let $\leq$ be a presumed subtype relation. A typed family of relations, $\mathcal{R}$, between Σ-algebras C and A is a *homomorphic relation between C and A for NomSig and* $\leq$ if and only if for all sets of typed identifiers X, whenever $\eta_1 \in \mathit{ENV}(X, C)$ and $\eta_2 \in \mathit{ENV}(X, A)$ are such that $\eta_1 \mathcal{R} \eta_2$, then for all $\mathtt{T} \in \mathit{TYPES}$, for all generic operation symbols $\mathtt{g} \in \mathit{GOP}$, and for all identifier lists $\vec{\mathtt{x}} : \vec{\mathtt{S}} \in X$ such that the nominal type of $\mathtt{g}(\vec{\mathtt{x}})$ is $\mathtt{T}$,

$$\mathcal{M}[\![\mathtt{g}(\vec{\mathtt{x}})]\!](C, \eta_1)\; \mathcal{R}_{\mathtt{T}}\; \mathcal{M}[\![\mathtt{g}(\vec{\mathtt{x}})]\!](A, \eta_2). \tag{6.2}$$

Formula 6.2 says that every possible result of $\mathcal{M}[\![\mathtt{g}(\vec{\mathtt{x}})]\!](C, \eta_1)$ is related by $\mathcal{R}_{\mathtt{T}}$ to some possible result of $\mathcal{M}[\![\mathtt{g}(\vec{\mathtt{x}})]\!](A, \eta_2)$.

We sometimes omit mention of the algebras, *NomSig*, and $\leq$ when they are not important or are clear from context.

In the definition above, the identifier list $\vec{\mathtt{x}}$ may be empty. Therefore, if $\mathcal{R}$ is a homomorphic relation between C and A, then for all nullary generic operation symbols (i.e., type symbols) $\mathtt{T} \in \mathit{GOP}$ such that the nominal type of `T()` is `TClass`, and for all environments η_1 and η_2,

$$\mathcal{M}[\![\mathtt{T()}]\!](C, \eta_1)\; \mathcal{R}_{\mathtt{TClass}}\; \mathcal{M}[\![\mathtt{T()}]\!](A, \eta_2). \tag{6.3}$$

This follows because the denotation of the expression `T()` does not depend on an environment, and furthermore if $\eta_1 \in \mathit{ENV}(\emptyset, C)$ and $\eta_2 \in \mathit{ENV}(\emptyset, A)$, then by definition $\eta_1 \mathcal{R} \eta_2$. So if there are nullary generic operation symbols in the domain of *NomSig*, then a homomorphic relation for *NomSig* and $\leq$ cannot be empty.

A trivial example of a homomorphic relation between an algebra A and itself is the typed family of equality relations on the carrier sets of A. A trivial example of a homomorphic relation between two algebras C and A is the typed family of relations each of which relates every object in the carrier set of C to every object in the carrier set of A. We require that simulation relations be V-identical and bistrict so that this "complete" homomorphic relation not a simulation relation.

Unless it preserves the meaning of the visible types, a homomorphic relation embodies a poor notion of "simulation." The following definition describes what we mean by preserving the meaning of the visible types. The definition makes sense because we have assumed that the same reduct that defines the visible types is used in each algebra.

Definition 6.1.4 (V-identical). Let $\Sigma = (\mathit{TYPES}, V, \mathit{TFUNS}, \mathit{OPS})$ be a signature. Let C and A be Σ-algebras. Then a typed family of binary relations $\mathcal{R}$ between the C and A is *V-identical* if and only if:

- for each type $\mathtt{T} \in \mathit{TYPES}$, if $q \mathcal{R}_{\mathtt{T}} r$ and either q or r is a proper instance of a visible type, then $q = r$,
- for each visible type $v \in V$, $\mathcal{R}_v$ contains the identity relation on A_v, and
- for each type $\mathtt{T} \in \mathit{TYPES}$ and for all visible types $v \in V$, if there is some proper element $q \in A_v$ such that $q \, \mathcal{R}_{\mathtt{T}} \, q$, then for all proper elements r in A_v, $r \, \mathcal{R}_{\mathtt{T}} \, r$.

The first condition says that distinct elements of the visible types cannot be related. The second condition says that at each visible type v, $\mathcal{R}_v$ is the identity on the carrier set of v. While at first glance it would seem better to prohibit $\mathcal{R}_{\mathtt{T}}$ from relating objects of a visible type v when $\mathtt{T} \neq v$, this restriction would prevent us from using simulation relations to prove that a visible type is a subtype of some other type. The third condition allows our proofs to work without such restrictions, although it is somewhat complex.

We also require that a simulation relation preserve the meaning of $\perp$, which represents nontermination and errors in our algebras.

Definition 6.1.5 (bistrict). A binary relation $\ll$ is *bistrict* if and only if $\perp \ll \perp$ and whenever $q \ll r$ and one of q or r is $\perp$, then so is the other.

Note that a bistrict relation cannot be empty. We call a typed family of relations $\mathcal{R}$ bistrict if each $\mathcal{R}_{\mathtt{T}}$ is bistrict.

As an example of a simulation relation, let *NomSig* be determined by the specification IPT (see Figure 2.2). Recall that for this specification, the presumed subtype relation $\leq$ is the smallest reflexive relation on the types of IPT such that `IntTriple` $\leq$ `IntPair`. Let A be the algebra of Figure 2.5. Let $\mathcal{R}$ be the smallest bistrict typed family of relations such that for all types $\mathtt{T}$, if $q \in A_{\mathtt{T}}$, then $q \, \mathcal{R}_{\mathtt{T}} \, q$, and for all $\langle q_1, q_2, q_3 \rangle, \langle q_1, q_2, q_4 \rangle \in A_{\mathtt{IntTriple}}$

and $\langle q_1, q_2 \rangle \in A_{\texttt{IntPair}}$,

$$\langle q_1, q_2, q_3 \rangle \;\; \mathcal{R}_{\texttt{IntPair}} \;\; \langle q_1, q_2 \rangle \tag{6.4}$$

$$\langle q_1, q_2 \rangle \;\; \mathcal{R}_{\texttt{IntPair}} \;\; \langle q_1, q_2, q_3 \rangle \tag{6.5}$$

$$\langle q_1, q_2, q_3 \rangle \;\; \mathcal{R}_{\texttt{IntPair}} \;\; \langle q_1, q_2, q_4 \rangle . \tag{6.6}$$

Then $\mathcal{R}$ is a simulation relation for *NomSig* and $\leq$. By construction, $\mathcal{R}$ is bistrict, V-identical, and relationships at type `IntTriple` hold at type `IntPair`. To show that $\mathcal{R}$ is a homomorphic relation for *NomSig* and $\leq$, let X and environments $\eta_1, \eta_2 \in ENV(X, A)$ be such that $\eta_1 \mathcal{R} \eta_2$. Suppose $\mathbf{g}(\vec{\mathbf{x}})$ is a generic invocation of some nominal type T. If none of the identifiers in $\vec{\mathbf{x}}$ have nominal type `IntPair`, then by construction $\eta_1(\vec{\mathbf{x}}) = \eta_2(\vec{\mathbf{x}})$ and thus $\mathcal{M}[\![\mathbf{g}(\vec{\mathbf{x}})]\!](A, \eta_1) = \mathcal{M}[\![\mathbf{g}(\vec{\mathbf{x}})]\!](A, \eta_2)$; since each $\mathcal{R}_{\mathtt{T}}$ contains the identity relation on $A_{\mathtt{T}}$, it follows that $\mathcal{M}[\![\mathbf{g}(\vec{\mathbf{x}})]\!](A, \eta_1) \; \mathcal{R}_{\mathtt{T}} \; \mathcal{M}[\![\mathbf{g}(\vec{\mathbf{x}})]\!](A, \eta_2)$. The only generic operation symbols that can take an argument of nominal type `IntPair` are **`first`** and **`second`**. Suppose $\mathbf{x}$ has nominal type `IntPair`, then since $\eta_1(\mathbf{x}) \mathcal{R}_{\texttt{IntPair}} \eta_2(\mathbf{x})$, the first and second components of $\eta_1(\mathbf{x})$ and $\eta_2(\mathbf{x})$ must be the same if they are proper objects. Therefore we have:

$$\mathcal{M}[\![\texttt{first(x)}]\!](A, \eta_1) \;=\; \mathcal{M}[\![\texttt{first(x)}]\!](A, \eta_2) \tag{6.7}$$

$$\mathcal{M}[\![\texttt{second(x)}]\!](A, \eta_1) \;=\; \mathcal{M}[\![\texttt{second(x)}]\!](A, \eta_2). \tag{6.8}$$

So this $\mathcal{R}$ is a homomorphic relation.

A homomorphic relation has a ***substitution property***: each generic invocation maps related arguments to related results (Formula 6.2). Moreover, the possible results of a generic invocation $\mathbf{g}(\vec{\mathbf{x}})$ are related by $\mathcal{R}_{\mathtt{T}}$, where the nominal type of $\mathbf{g}(\vec{\mathbf{x}})$ is T. This substitution property is the key to our inductive proof that simulation is preserved by all type-safe NOAL expressions.

The substitution property is similar to the defining property of Nipkow's simulation relations [Nip86]. The difference is that we provide for generic invocation mechanisms and allow objects of one type to be related to objects of another type.

A substitution property is also used to define homomorphisms between multisorted algebras [Gra79, Page 35]. Our homomorphic relations differ from the usual homomorphisms between multisorted algebras [EM85] in two ways: they are relations instead of functions, and they may relate objects of one type to objects of another type. Nevertheless, it is appropriate to call homomorphic relations "homomorphic," because they have a substitution property and they can be composed. (The composition of homomorphic relations is just the usual product of relations at each type.)

Simulation relations are also similar to, but more general than, the coercer functions that form the basis of Bruce and Wegner's definitions of subtype relations [BW87].

Homomorphic relations also resemble the "logical relations" used in the study of the lambda calculus [Sta85] [Mit86]. An important property of logical relations is captured by the so-called fundamental theorem of logical relations [Sta85]. In the next section we state and prove our version of the fundamental theorem (which is Theorem 6.2.3).

6.2 Simulation as a Criterion for Imitation

In this section we show that simulation is preserved by all type-safe NOAL expressions. That is, the substitution property holds for NOAL expressions and programs as well as generic invocations. This allows us to conclude that if $\mathcal{R}$ is a simulation relation for *NomSig* and $\leq$ and $\mathcal{R}$ relates two environments, then the first environment imitates the second with respect to the set of type-safe NOAL programs over *NomSig* and $\leq$. This result is used in the next section in our technique for showing that a binary relation on types is a subtype relation.

Our strategy is as follows. We first show that simulation is preserved by all NOAL expressions. Since simulation relations are V-identical and bistrict, and since type-safe programs have visible type, it follows that simulation can be used as a criterion for imitation. To show the desired result for NOAL expressions, we first assume that the denotations of each free function identifier in the two algebras are related by $\mathcal{R}$ (in a way we describe below). We then show (in Appendix C) that the two meanings, one for each algebra, of a system of recursively defined function identifiers are properly related.

Since NOAL "functions" can be nondeterministic, their denotations are operations (i.e., set-valued functions). We compare operations by analogy to the definition of logical relations [Sta85] [Mit86]. That is, if $\mathcal{R}$ is a typed family of relations, then we extend $\mathcal{R}$ to the signatures of NOAL function identifiers as follows:

$$\mathcal{R}_{\vec{\tau}\to\sigma} \stackrel{\text{def}}{=} \{(f_1, f_2) \mid \vec{q}\,\mathcal{R}_{\vec{\tau}}\,\vec{r} \Rightarrow f_1(\vec{q})\,\mathcal{R}_{\sigma}\,f_2(\vec{r})\}\,. \tag{6.9}$$

That is, for all operations f_1 and f_2, f_1 is related by $\mathcal{R}_{\vec{\tau}\to\sigma}$ to f_2 if and only if whenever $\vec{q}\,\mathcal{R}_{\vec{\tau}}\,\vec{r}$, then for every $q' \in f_1(\vec{q})$, there is some $r' \in f_2(\vec{r})$ such that $q'\,\mathcal{R}_\sigma\,r'$. Notice that this extension of $\mathcal{R}$ preserves the substitution property of simulation relations. Since operations are not first-class objects in NOAL, we need not be concerned that this extension has all the properties of a simulation relation at each function signature.

To deal with NOAL expressions that have free function identifiers, we allow environments to map typed function identifiers to operations (i.e., to relations that denote recursively defined NOAL functions).

For brevity, throughout the rest of this section we establish the following conventions.

$\Sigma = (SORTS, TYPES, V, TFUNS, OPS)$ is a signature.

A and B are Σ-algebras.

NomSig is a nominal signature map.

GOP is the domain of *NomSig*, a set of generic operation symbols.

$\leq$ is a binary relation on type symbols.

Recall that *NomSig* and $\leq$ can be used to determine the nominal type of a NOAL program or expression, if one exists.

6.2.1 The Substitution Property for NOAL Expressions

Informally, the following lemma says that simulation is preserved by NOAL expressions if it is preserved by each recursively defined function.

Lemma 6.2.1. Let X be a set of typed identifiers and function identifiers. Let $\eta_1 \in ENV(X, A)$ and $\eta_2 \in ENV(X, B)$ be environments.

If $\mathcal{R}$ is a simulation relation between A and B for *NomSig* and $\leq$ and if $\eta_1\,\mathcal{R}\,\eta_2$, then for all types T and for all recursion-free NOAL expressions E of nominal type T whose free identifiers and function identifiers are a subset of X,

$$\mathcal{M}[\![E]\!](A, \eta_1)\,\mathcal{R}_{\mathtt{T}}\,\mathcal{M}[\![E]\!](B, \eta_2).$$

Proof: (by induction on the structure of expressions).

For the basis, we suppose that the expression is either an identifier, `bottom[T]`, or the invocation of a nullary generic operation symbol. If the expression is an identifier, then the result follows from $\eta_1 \mathcal{R} \eta_2$. If the expression is `bottom[T]` for some type T, then the result follows from the bistrictness of $\mathcal{R}_{\mathtt{T}}$. If the expression is `g()` for some nullary generic operation symbol g of nominal signature $\rightarrow$ T, then since $\eta_1 \mathcal{R} \eta_2$, by definition of a homomorphic relation we have

$$\mathcal{M}[\![\mathtt{g()}]\!](A, \eta_1)\, \mathcal{R}_{\mathtt{T}}\, \mathcal{M}[\![\mathtt{g()}]\!](B, \eta_2).$$

For the inductive step we assume that if $\eta_1 \mathcal{R} \eta_2$, then the denotation of each subexpression of nominal type T in the environment η_1 is related by $\mathcal{R}_{\mathtt{T}}$ to the denotation of the same subexpression in the environment η_2. There are several cases (see Figures 3.1 and 3.2).

- Suppose the expression is $\mathtt{g}(\vec{E})$ and $\vec{E}$ is not empty. Since this expression has a nominal type, by the type inference rules it must be that $\vec{\mathtt{S}} \rightarrow \mathtt{T} \in \mathit{NomSig}(\mathtt{g})$, $\vec{E}$ has nominal type $\vec{\sigma}$, $\sigma_1 = \mathtt{S}_1$, and for i from 2 to n, $\sigma_i \leq \mathtt{S}_i$. Let $\vec{q} \in \mathcal{M}[\![\vec{E}]\!](A, \eta_1)$ be given. By the inductive hypothesis, there is some $\vec{r} \in \mathcal{M}[\![\vec{E}]\!](B, \eta_2)$ such that $\vec{q}\, \mathcal{R}_{\vec{\sigma}}\, \vec{r}$. Since $\sigma_1 = \mathtt{S}_1$, $\mathcal{R}_{\sigma_1} = \mathcal{R}_{\mathtt{S}_1}$, and since for i from 2 to n, $\sigma_i \leq \mathtt{S}_i$, $\mathcal{R}_{\sigma_i} \subseteq \mathcal{R}_{\mathtt{S}_i}$. Therefore $\mathcal{R}_{\vec{\sigma}} \subseteq \mathcal{R}_{\vec{\mathtt{S}}}$ and so $\vec{q}\, \mathcal{R}_{\vec{\mathtt{S}}}\, \vec{r}$. Let $\vec{\mathtt{z}} : \vec{\mathtt{S}}$ be a list of typed identifiers, none of which occurs free in $\vec{E}$. Since $\vec{q}\, \mathcal{R}_{\vec{\mathtt{S}}}\, \vec{r}$, if we bind $\vec{q}$ to $\vec{\mathtt{z}}$ in η_1 (written $\eta_1[\vec{q}/\vec{\mathtt{z}}]$) and we bind $\vec{r}$ to $\vec{\mathtt{z}}$ in η_2, then $(\eta_1[\vec{q}/\vec{\mathtt{z}}])\, \mathcal{R}\, (\eta_2[\vec{r}/\vec{\mathtt{z}}])$. Since $\mathcal{R}$ is a homomorphic relation, it follows that

$$\mathcal{M}[\![\mathtt{g}(\vec{\mathtt{z}})]\!](A, \eta_1[\vec{q}/\vec{\mathtt{z}}])\, \mathcal{R}_{\mathtt{T}}\, \mathcal{M}[\![\mathtt{g}(\vec{\mathtt{z}})]\!](B, \eta_2[\vec{r}/\vec{\mathtt{z}}]). \tag{6.10}$$

  Furthermore, from the definition of NOAL one can show that

$$\mathcal{M}[\![\mathtt{g}(\vec{E})]\!](A, \eta_1) = \bigcup_{\vec{q} \in \mathcal{M}[\![\vec{E}]\!](A, \eta_1)} \mathcal{M}[\![\mathtt{g}(\vec{\mathtt{z}})]\!](A, \eta_1[\vec{q}/\vec{\mathtt{z}}]). \tag{6.11}$$

  Therefore, for every possible result $q \in \mathcal{M}[\![\mathtt{g}(\vec{E})]\!](A, \eta_1)$ there is some $r \in \mathcal{M}[\![\mathtt{g}(\vec{E})]\!](B, \eta_2)$ such that $q\, \mathcal{R}_{\mathtt{T}}\, r$.

- Suppose the expression is $\mathtt{f}(\vec{E})$ and f is a function identifier of nominal signature $\vec{\mathtt{S}} \rightarrow \mathtt{T}$. Since this expression has a nominal type, by the type inference rules it must be that $\vec{E}$ has nominal type $\vec{\sigma}$ and $\vec{\sigma} \leq \vec{\mathtt{S}}$. Let $\vec{q} \in \mathcal{M}[\![\vec{E}]\!](A, \eta_1)$ be given. By the inductive hypothesis, there is some $\vec{r} \in \mathcal{M}[\![\vec{E}]\!](B, \eta_2)$ such that $\vec{q}\, \mathcal{R}_{\vec{\sigma}}\, \vec{r}$. Since $\vec{\sigma} \leq \vec{\mathtt{S}}$, $\mathcal{R}_{\vec{\sigma}} \subseteq \mathcal{R}_{\vec{\mathtt{S}}}$ and so $\vec{q}\, \mathcal{R}_{\vec{\mathtt{S}}}\, \vec{r}$. Since $\eta_1\, \mathcal{R}\, \eta_2$, $\eta_1(\mathtt{f})\, \mathcal{R}_{\vec{\mathtt{S}} \rightarrow \mathtt{T}}\, \eta_2(\mathtt{f})$, and thus $(\eta_1(\mathtt{f}))(\vec{q})\ \mathcal{R}_{\mathtt{T}}\ (\eta_2(\mathtt{f}))(\vec{r})$. So, by definition of NOAL, for every possible result $q \in \mathcal{M}[\![\mathtt{f}(\vec{E})]\!](A, \eta_1)$ there is some $r \in \mathcal{M}[\![\mathtt{f}(\vec{E})]\!](B, \eta_2)$ such that $q\,\mathcal{R}_{\mathtt{T}}\,r$.

- Suppose the expression is $(\mathtt{fun}(\vec{\mathtt{x}} : \vec{\mathtt{S}})E_0)(\vec{E})$ and that the nominal type of the entire expression is T. Let $\vec{q} \in \mathcal{M}[\![\vec{E}]\!](A, \eta_1)$ be given. By the inductive hypothesis, there is some $\vec{r} \in \mathcal{M}[\![\vec{E}]\!](B, \eta_2)$ such that $\vec{q}\,\mathcal{R}_{\vec{\sigma}}\,\vec{r}$, where $\vec{\sigma}$ is the nominal type of $\vec{E}$. Since the expression has a nominal type, by the type inference rules for NOAL it must be that $\vec{\sigma} \leq \vec{\mathtt{S}}$; thus $\vec{q}\,\mathcal{R}_{\vec{\mathtt{S}}}\,\vec{r}$. It follows that if we bind $\vec{q}$ to $\vec{\mathtt{x}}$ in η_1 and $\vec{r}$ to $\vec{\mathtt{x}}$ in η_2, then $(\eta_1[\vec{q}/\vec{\mathtt{x}}])\ \mathcal{R}\ (\eta_2[\vec{r}/\vec{\mathtt{x}}])$; thus the result follows by the inductive hypothesis (applied to E_0).

- Suppose the expression is **if** E_1 **then** E_2 **else** E_3 **fi**. Since Bool is a visible type and $\mathcal{R}$ is V-identical, the possible results from E_1 in η_1 are a subset of those possible in η_2. Therefore the result follows from the inductive hypothesis applied to E_2 and E_3.

- Suppose the expression is $E_1 \,[\,]\, E_2$. The possible results of this expression are the union of those from E_1 and E_2. Since the expression has a nominal type, the types of both E_1 and E_2 must be the same, say T. By the inductive hypothesis, for every possible result q of E_1 in the environment η_1 there is some possible result r from E_1 in the environment η_2 such that $q\,\mathcal{R}_{\mathtt{T}}\,r$; similarly for E_2. Hence the result follows.

- Suppose the expression is $E_1 \bigtriangledown E_2$ and has type T. The possible results of this expression are the union of those from E_1 and E_2, except that $\perp$ appears only if it is a possible result of both. This is the same as the previous case, except that we have to be careful about $\perp$. Suppose, for the sake of contradiction, that there was some q in $\mathcal{M}[\![E_1 \bigtriangledown E_2]\!](A, \eta_1)$ such that q is not related by $\mathcal{R}_{\mathtt{T}}$ to some element of $\mathcal{M}[\![E_1 \bigtriangledown E_2]\!](B, \eta_2)$. By the previous case, if $\mathcal{M}[\![E_1 [\,] E_2]\!](B, \eta_2)$ is the same as $\mathcal{M}[\![E_1 \bigtriangledown E_2]\!](B, \eta_2)$, then this would be a contradiction; so we can assume that $\mathcal{M}[\![E_1 \bigtriangledown E_2]\!](B, \eta_2)$ does not include $\perp$ and $q\,\mathcal{R}_{\mathtt{T}}\,\perp$. Since $\mathcal{R}_{\mathtt{T}}$ is bistrict, it must be that $q = \perp$. Furthermore, by definition of $\bigtriangledown$, since $\mathcal{M}[\![E_1 \bigtriangledown E_2]\!](B, \eta_2)$ does not include $\perp$, it must be that either E_1 or E_2 is guaranteed to terminate in B and η_2. Without loss of generality, suppose $\perp \notin \mathcal{M}[\![E_1]\!](B, \eta_2)$.

Then $\perp \notin \mathcal{M}[\![E_1]\!](A, \eta_1)$, since $\mathcal{R}$ is bistrict and by the inductive hypothesis $\mathcal{M}[\![E_1]\!](A, \eta_1) \; \mathcal{R}_{\mathtt{T}} \; \mathcal{M}[\![E_1]\!](B, \eta_2)$. But then by definition of NOAL's angelic choice operator, $\perp$ is not in $\mathcal{M}[\![E_1 \bigtriangledown E_2]\!](A, \eta_1)$. This contradicts the assumption that $q \in \mathcal{M}[\![E_1 \bigtriangledown E_2]\!](A, \eta_1)$, since $q = \perp$. Hence the result follows.

- If the expression is `isDef?(`E_1`)`, then the result follows directly from the inductive hypothesis applied to E_1 and the bistrictness of $\mathcal{R}$.

∎

The proof of the above lemma is the source of our requirement that simulation relationships at a subtype also hold at each supertype. This property guarantees that expressions related at a subtype are also related when we exploit inclusion polymorphism. For example, if E has nominal type `S`, `S` is a subtype of `T`, and the function identifier `f` has nominal signature `T` $\rightarrow$ `U`, then the expression `f(`E`)` is type-safe; furthermore, if the meanings of E in η_1 and η_2 are related at type `S` and if the meanings of `f` are also related at type `T` $\rightarrow$ `U`, then by this property the arguments are related at the nominal argument type `T`, and thus the results are related at the nominal result type `U`.

6.2.2 The Substitution Property for NOAL Programs

To show that the substitution property holds for NOAL programs we need to show that simulation is preserved by recursively-defined NOAL functions. The proof is involved because of NOAL's erratic and angelic choice expressions and has therefore been relegated to Appendix C. However, the idea of the proof can be stated as follows. To deal with functions that use only erratic choice one uses a family of approximations, each of which is deterministic and that together cover the choices available to functions that use erratic choice. To deal with angelic choice one first rewrites the functions, replacing angelic with erratic choices and uses the limit of the erratic choice approximations as a first approximation. Then one expands recursive calls in-line, obtaining a series of approximations that use angelic choice for deeper and deeper recursions. At each stage of approximation, simulation is preserved. We also show that simulation is preserved by the various limit operators. This series of lemmas culminates in the following result.

Lemma 6.2.2. Let

$$\begin{array}{l}\texttt{fun } \mathbf{f}_1(\vec{\mathbf{x}_1} : \vec{\mathbf{S}_1}) : \mathbf{T}_1 = E_1; \\ \vdots \\ \texttt{fun } \mathbf{f}_m(\vec{\mathbf{x}_m} : \vec{\mathbf{S}_m}) : \mathbf{T}_m = E_m\end{array}$$

be a mutually recursive system of NOAL function definitions.

Suppose $\mathcal{R}$ is a simulation relation between algebras A and B for *NomSig* and $\leq$. Then for each j from 1 to m,

$$\mathcal{F}(A)[\![\mathbf{f}_j]\!] \, \mathcal{R}_{\vec{\mathbf{S}_j} \to \mathbf{T}_j} \, \mathcal{F}(B)[\![\mathbf{f}_j]\!]. \tag{6.12}$$

Proof: See Appendix C. ∎

Using the above lemma, we can prove that simulation is a valid criterion for imitation.

Theorem 6.2.3. Let Σ be a signature. Let A and B be Σ-algebras. Let *NomSig* be a nominal signature map. Let $\leq$ be a binary relation on type symbols.

Suppose that $\mathcal{R}$ is a simulation relation between A and B for *NomSig* and $\leq$. Then for all sets of typed identifiers X, for all environments $\eta_A \in ENV(X, A)$, and for all environments $\eta_B \in ENV(X, B)$, if $\eta_A \, \mathcal{R} \, \eta_B$, then (A, η_A) imitates (B, η_B) with respect to the set of all type-safe NOAL programs over *NomSig* and $\leq$.

Proof: Suppose that $\mathcal{R}$ is a simulation relation between A and B for *NomSig* and $\leq$. Let X be a set of typed identifiers and let $\eta_1 \in ENV(X, A)$ and $\eta_2 \in ENV(X, B)$ be such that $\eta_1 \, \mathcal{R} \, \eta_2$. Let P be a type-safe NOAL program over *NomSig* and $\leq$. In general P has the form

$$\begin{array}{l}\texttt{fun } \mathbf{f}_1(\vec{\mathbf{z}_1} : \vec{\mathbf{S}_1}) : \mathbf{T}_1 = E_1; \\ \vdots \\ \texttt{fun } \mathbf{f}_m(\vec{\mathbf{z}_m} : \vec{\mathbf{S}_m}) : \mathbf{T}_m = E_m; \\ E.\end{array}$$

Since P is type-safe, P has some nominal type, say $\mathbf{T}$. Let Z be the set of typed function identifiers that contains the $\mathbf{f}_j$ with their nominal signatures. Let $\eta_1' \in ENV(Z \cup X, A)$ and $\eta_2' \in ENV(Z \cup X, B)$ be defined so that for all $\mathbf{x} \in X$, $\eta_1'(\mathbf{x}) = \eta_1(\mathbf{x})$, $\eta_2'(\mathbf{x}) = \eta_2(\mathbf{x})$ and for all $\mathbf{f}_j \in Z$, $\eta_1'(\mathbf{f}_j)$ is the denotation of $\mathbf{f}_j$ in A and $\eta_2'(\mathbf{f}_j)$ is the denotation of $\mathbf{f}_j$ in B. By Lemma 6.2.2, $\eta_1' \, \mathcal{R} \, \eta_2'$, since the denotations of recursively defined functions are related by $\mathcal{R}$. So by Lemma 6.2.1,

$$\mathcal{M}[\![E]\!](A, \eta_1') \, \mathcal{R}_{\mathbf{T}} \, \mathcal{M}[\![E]\!](B, \eta_2'). \tag{6.13}$$

Therefore, by the semantics of NOAL programs,

$$\mathcal{M}[\![P]\!](A,\eta_1)\ \mathcal{R}_{\mathtt{T}}\ \mathcal{M}[\![P]\!](B,\eta_2). \tag{6.14}$$

Recall that this means that for each $q \in \mathcal{M}[\![P]\!](A,\eta_1)$, there is some $r \in \mathcal{M}[\![P]\!](B,\eta_2)$ such that $q\,\mathcal{R}_{\mathtt{T}}\,r$. Since P is a program, its nominal type must be a visible type; that is, $\mathtt{T} \in V$. Since $\mathcal{R}$ is V-identical, for each $q \in \mathcal{M}[\![P]\!](A,\eta_1)$, there is some $r \in \mathcal{M}[\![P]\!](B,\eta_2)$ such that $q = r$; that is,

$$\mathcal{M}[\![P]\!](A,\eta_1) \subseteq \mathcal{M}[\![P]\!](B,\eta_2). \tag{6.15}$$

Therefore (A,η_1) imitates (B,η_2) with respect to the set of type-safe NOAL programs over *NomSig* and $\leq$. So by definition, $\mathcal{R}$ is an imitation relation. ∎

The significance of the above theorem is that simulation relations can be used to prove that a binary relation on types is a subtype relation with respect to the set of type-safe NOAL programs, as discussed in the next section.

6.3 Proving Subtype Relations using Simulation Relations

In this section we describe our technique of proving subtype relations using a simulation relation.

The idea is to find simulation relations that witness a subtype relation in the sense that they relate every instance of a subtype to some instance of its supertypes.

Definition 6.3.1 (witnesses). Let Σ be a signature. Let C and A be Σ-algebras. Let $\leq$ be a binary relation on types. A typed family of relations $\mathcal{R}$ between C and A ***witnesses*** $\leq$ if and only if for all types $\mathtt{S}$ and $\mathtt{T}$,

$$(\mathtt{S} \leq \mathtt{T}) \Rightarrow (C_{\mathtt{S}}\ \mathcal{R}_{\mathtt{T}}\ A_{\mathtt{T}}). \tag{6.16}$$

To prove that $\leq$ is a subtype relation on the types of *SPEC* with respect to type-safe NOAL programs, one must find simulation relations that witness $\leq$ for each *SPEC*-algebra.

Theorem 6.3.2. Let Σ be a signature. Let *SPEC* be a set of Σ-algebras. Let *NomSig* be a nominal signature map. Let $\leq$ be a binary relation on types.

If for every $C \in \mathit{SPEC}$ there is some $A \in \mathit{SPEC}$ and some simulation relation $\mathcal{R}$ between C and A for *NomSig* and $\leq$ such that $\mathcal{R}$ witnesses $\leq$, then $\leq$ is a subtype relation on the types of *SPEC* with respect to the set of type-safe NOAL programs over *NomSig* and $\leq$.

Proof: Let $C \in \mathit{SPEC}$ be given. By hypothesis, there is some $A \in \mathit{SPEC}$ and some simulation relation $\mathcal{R}$ between C and A for *NomSig* and $\leq$ such that $\mathcal{R}$ witnesses $\leq$. Let X be a set of typed identifiers. Let $\eta_C \in \mathit{ENV}(X, C)$ be such that η_C obeys $\leq$.

Since $\mathcal{R}$ witnesses $\leq$ we can build a nominal environment $\eta_A \in \mathit{ENV}(X, A)$ such that $\eta_C \,\mathcal{R}\, \eta_A$. That is, for each $\mathbf{x} : \mathtt{T} \in X$, there is some type $\mathtt{S} \leq \mathtt{T}$ such that $\eta_C(\mathbf{x}) \in C_{\mathtt{S}}$. Since $\mathcal{R}$ witnesses $\leq$, there is some $r \in A_{\mathtt{T}}$ such that $\eta_C(\mathbf{x}) \,\mathcal{R}\, r$. So let $\eta_A(\mathbf{x}) \overset{\text{def}}{=} r$.

Since $\eta_C \,\mathcal{R}\, \eta_A$, by Theorem 6.2.3, (C, η_C) imitates (A, η_A) with respect to the set of type-safe NOAL programs over *NomSig* and $\leq$. So by definition, $\leq$ is a subtype relation. ∎

The above technique for proving subtype relations may not be complete. That is, although finding simulation relations that witness $\leq$ is sufficient to prove that $\leq$ is a subtype relation, we do not know whether this condition is necessary for $\leq$ to be a subtype relation. A simpler problem, which, if it can be answered in the affirmative, would imply the completeness of our technique for proving subtype relations, is the following: if (C, η_C) imitates (A, η_A) with respect to the set of type-safe NOAL programs over *NomSig* and $\leq$, is there a simulation relation $\mathcal{R}$ between C and A for *NomSig* and $\leq$ such that $\eta_C \,\mathcal{R}\, \eta_A$? However, this simpler problem is not equivalent to the problem of whether our technique for proving subtype relations is complete.

In the rest of this section we look at some ways to make our proof technique more practical.

6.3.1 Modularity in Proofs of Subtype Relations

In this subsection we discuss the problem of proving subtype relationships instead of subtype relations.

Our formal treatment of subtyping deals with subtype relations instead of subtype relationships. So formally, we cannot prove that one type is a subtype of another, but only that a particular binary relation on types is a subtype relation. This is less modular than one would like, because it requires one to look at all the types in a program at once instead of looking at individual subtype relationships.

To attack the modularity problem directly one would have to find conditions on type specifications and the set of observations that would allow individual subtype relationships to be checked. Another way to attack this problem would be to consider adding types one at a time to a specification and to consider the question of what conditions on the new type specification will ensure that a subtype relation on the old type specification is still a subtype relation on the new type specification. We leave these approaches as problems for future work.

One reason for leaving this problem for future work is the difficulty of giving a general definition of subtype relationships instead of subtype relations. One way to define subtype relationships would be to show that for each specification there is a largest subtype relation, and then to say that S is a subtype of T if they are related by this largest relation. The binary relations on a set of type symbols can be partially ordered by set inclusion ($\subseteq$). If $\leq_1 \subseteq \leq_2$, then we say that $\leq_2$ is *larger than* $\leq_1$. So $\leq$ is the *largest* subtype relation on the types of a specification if it is larger than all other subtype relations on that specification's types. However, we will show by an example below that with respect to an arbitrary set of observations there may be no largest subtype relation for a given specification.

Another problem with largest subtype relations is that the set of observations defined by type-safe programs changes when the presumed subtype relation changes. For example, the set of type-safe NOAL programs is determined, in part, by the presumed subtype relation. The same is true in Trellis/Owl. We do not know whether there are example specifications where there are two incomparable binary relations on types such that each is a subtype relation with respect to the corresponding set of type-safe programs.

On the other hand, the lack of a largest subtype relation is not terribly important to programmers. One normally writes a type specification with some subtype relation in mind, since one has to choose the names of instance operations and specifications carefully to make the desired subtype relationships hold. So programmers should be more interested in whether a particular binary relation on types is a subtype relation than in finding all the subtype relations for a given specification.

In the rest of this section we present an example with two incomparable subtype relations and no subtype relation that is larger than both. Our example involves two types, T1 and T2. Since our specification language cannot handle this example, consider the algebra A presented in Figure 6.1 as the only algebra in the semantics of a specification we will call T12.

Let $\leq_1$ be the smallest reflexive relation on the types of T12 such that $\mathtt{T1} \leq_1 \mathtt{T2}$. Let $\leq_2$ be the smallest reflexive relation on the types of T12 such that $\mathtt{T2} \leq_2 \mathtt{T1}$. We claim that $\leq_1$ and $\leq_2$ are subtype relations on the types of T12 with respect to the set of observations OBS described by the NOAL program `g(x2, x1)`, where $\mathtt{x1} : \mathtt{T1}$ and $\mathtt{x2} : \mathtt{T2}$.

Lemma 6.3.3. Let $\leq_1$ and OBS be as described above. Then the relation $\leq_1$ is a subtype relation on the types of T12 with respect to OBS.

Proof: Let X be a set of typed identifiers, indexed by the types of T12. Let $\eta_A \in ENV(X, A)$ be an environment that obeys $\leq_1$. If $\{\mathtt{x1} : \mathtt{T1}, \mathtt{x2} : \mathtt{T2}\} \not\subseteq X$, then by definition, (A, η_A) imitates (A, η) with respect to OBS, for all nominal environments $\eta \in ENV(X, A)$. If $\{\mathtt{x1} : \mathtt{T1}, \mathtt{x2} : \mathtt{T2}\} \subseteq X$, then let $\eta_2 \in ENV(X, A)$ be defined so that for all $\mathbf{x} : \mathtt{T1} \in X$, $\eta_2(\mathbf{x}) \stackrel{\text{def}}{=} a$, for all $\mathbf{x} : \mathtt{T2} \in X$, $\eta_2(\mathbf{x}) \stackrel{\text{def}}{=} b$, and for all other identifiers, $\eta_2(\mathbf{x}) = \eta_A(\mathbf{x})$. Note that η_2 is a nominal environment. The only observation in OBS is the program above. Note that

$$\mathcal{M}[\![\mathbf{g}(\mathbf{x2}, \mathbf{x1})]\!](A, \eta_A) = \{\mathit{false}\} \tag{6.17}$$

$$\mathcal{M}[\![\mathbf{g}(\mathbf{x2}, \mathbf{x1})]\!](A, \eta_2) = \{\mathit{false}\}. \tag{6.18}$$

The first equation holds because the generic invocation mechanism either calls $\mathbf{g}_{\mathbf{T2},\mathbf{T1}\rightarrow\mathbf{Bool}}$ or $\mathbf{g}_{\mathbf{T1},\mathbf{T1}\rightarrow\mathbf{Bool}}$. Therefore, (A, η_A) imitates (A, η_2) with respect to OBS. Since for all environments η_A this construction produces a nominal algebra-environment pair (A, η_2), by definition $\leq$ is a subtype relation. ∎

Lemma 6.3.4. Let $\leq_2$ and OBS be as described above. Then the relation $\leq_2$ is a subtype relation on the types of T12 with respect to OBS.

Proof: Let $\eta_A \in ENV(X, A)$ be an environment that obeys $\leq_2$. If $\{\mathtt{x1} : \mathtt{T1}, \mathtt{x2} : \mathtt{T2}\} \subseteq X$, then we can define a nominal environment $\eta_2 \in ENV(X, A)$ as in the previous

Figure 6.1: The T12-algebra, A.

Carrier Sets

$$\begin{array}{lcl}
A_{\mathtt{T1}} & \overset{\text{def}}{=} & \{\bot, a\} \\
A_{\mathtt{T1Class}} & \overset{\text{def}}{=} & \{\bot, \mathit{T1}\} \\
A_{\mathtt{T2}} & \overset{\text{def}}{=} & \{\bot, b\} \\
A_{\mathtt{T2Class}} & \overset{\text{def}}{=} & \{\bot, \mathit{T2}\} \\
A_{\mathtt{Bool}} & \overset{\text{def}}{=} & \{\bot, \mathit{true}, \mathit{false}\} \\
\vdots & &
\end{array}$$

Trait Functions

$$\begin{array}{lcl}
A_{\mathrm{T1}}() & \overset{\text{def}}{=} & \mathit{T1} \\
A_{\mathrm{a}}() & \overset{\text{def}}{=} & a \\
A_{\mathrm{T2}}() & \overset{\text{def}}{=} & \mathit{T2} \\
A_{\mathrm{b}}() & \overset{\text{def}}{=} & b \\
A_{\mathrm{Bool}}() & \overset{\text{def}}{=} & \mathit{Bool} \\
\vdots & &
\end{array}$$

Operations

$$\begin{array}{lcl}
A_{\mathtt{T1\_\to T1Class}}() & \overset{\text{def}}{=} & \{\mathit{T1}\} \\
A_{\mathtt{newT1Class\to T1}}(\mathit{T1}) & \overset{\text{def}}{=} & \{a\} \\
A_{\mathtt{gT1,T1\to Bool}}(x_1, x_2) & \overset{\text{def}}{=} & \{\mathit{false}\} \\
A_{\mathtt{gT1,T2\to Bool}}(x, y) & \overset{\text{def}}{=} & \{\mathit{true}\} \\
A_{\mathtt{T2\_\to T2Class}}() & \overset{\text{def}}{=} & \{\mathit{T2}\} \\
A_{\mathtt{newT2Class\to T2}}(\mathit{T2}) & \overset{\text{def}}{=} & \{b\} \\
A_{\mathtt{gT2,T2\to Bool}}(y_1, y_2) & \overset{\text{def}}{=} & \{\mathit{false}\} \\
A_{\mathtt{gT2,T1\to Bool}}(y, x) & \overset{\text{def}}{=} & \{\mathit{false}\} \\
A_{\mathtt{Bool\_\to BoolClass}}() & \overset{\text{def}}{=} & \{\mathit{Bool}\} \\
\vdots & &
\end{array}$$

proof. Again

$$\mathcal{M}[\![\mathtt{g(x2,x1)}]\!](A,\eta_A) = \{\mathit{false}\} \tag{6.19}$$

$$\mathcal{M}[\![\mathtt{g(x2,x1)}]\!](A,\eta_2) = \{\mathit{false}\}, \tag{6.20}$$

where the first equation holds because the generic invocation mechanism either calls $g_{\mathtt{T2,T1\rightarrow Bool}}$ or $g_{\mathtt{T2,T2\rightarrow Bool}}$. ∎

However, the union $\leq_1$ and $\leq_2$ is not a subtype relation with respect to *OBS*. To see this, let $X = \{\mathtt{x1} : \mathtt{T1}, \mathtt{x2} : \mathtt{T2}\}$, and consider the environment $\eta_3 \in ENV(X, A)$ defined such that $\eta_3(\mathtt{x1}) \stackrel{\text{def}}{=} b$ and $\eta_3(\mathtt{x2}) \stackrel{\text{def}}{=} a$. Then η_3 obeys $\leq_1 \cup \leq_2$, but

$$\mathcal{M}[\![\mathtt{g(x2,x1)}]\!](C,\eta_3) = \{\mathit{true}\}, \tag{6.21}$$

although this cannot happen in a nominal environment.

As we show in the following lemma, there can be no subtype relation that contains $\leq_1 \cup \leq_2$, since every subset of a subtype relation is also a subtype relation.

Lemma 6.3.5. Let Σ be a signature and let *SPEC* be a nonempty set of Σ-algebras. Let *OBS* be a set of observations. Let $\leq$ be a binary relation on types.

If $\leq$ is a subtype relation on the types of *SPEC* with respect to *OBS*, then every subset of $\leq$ is a subtype relation on the types of *SPEC* with respect to *OBS*.

Proof: Suppose $\leq'$ is a subset of $\leq$. Let $C \in SPEC$ be given. Let X be a set of typed identifiers, and let $\eta_C \in ENV(X, C)$ be such that η_C obeys $\leq'$. Since $\leq' \subseteq \leq$, η_C obeys $\leq$. So by definition of subtype relation, there is some $A \in SPEC$ and some nominal environment $\eta_A \in ENV(X, A)$ such that (C, η_C) imitates (A, η_A) with respect to *OBS*. Since the same algebra A can be used for all environments over C, $\leq'$ is a subtype relation. ∎

Therefore there is no largest subtype relation on the types of T12 with respect to the set of observations described by the NOAL program `g(x1,x2)`.

One might object that the set of observations for the above example is so small. For example, if the program `g(x1,x2)` were included in the set of observations, then neither $\leq_1$ nor $\leq_2$ would be a subtype relation with respect to this enlarged set of observations. We leave it as an open problem whether there is some set of conditions on specifications and observations that ensures that there are largest subtype relations.

In general there may not be a largest simulation relation between two algebras for a given nominal signature map and presumed subtype relation. We could show this by adapting one of Nipkow's examples that has two incomparable simulation relations whose union is not a simulation relation [Nip87, Page 52].

6.3.2 Using Universal Models in Proving Subtype Relations

If one wants to prove that a binary relation is a subtype relation, it would be convenient to just work with the abstract values of types [Hoa72] instead of all algebraic models. Many specifications have algebraic models whose carrier sets can be thought of as the set of abstract values of the corresponding types. Such specifications are said to have universal models.

Intuitively, a universal model of a type specification is maximally nondeterministic and therefore exhibits all the behavior allowed by that specification. That is, all models of the specification are implementations of the universal model [Hes88]. Because of this property, a universal model of a specification, if it exists, is also as "abstract" as possible. Therefore, if a specification has a universal model we can identify the carrier set of each type in the universal model with the abstract values of each type. Furthermore, we will show that if we have a simulation relation between a universal model and itself that witnesses $\leq$ then we can conclude that $\leq$ is a subtype relation.

Instead of defining universal models using sorted homomorphisms on algebras, the following definition uses the imitates relation, which is more convenient for us. However, since sorted homomorphisms do not map objects of one type to objects of another type, the following definition does require that every environment imitates some environment of the universal model where each identifier is mapped to an object of the same type.

Definition 6.3.6 (universal model). Let Σ be a signature and let *SPEC* be a set of Σ-algebras. Let *OBS* be a set of observations. Then A is a ***universal model of SPEC with respect to OBS*** if and only if for every algebra $C \in SPEC$, for all sets of typed identifiers X, and for all environments $\eta_C \in ENV(X, C)$, there is some environment $\eta_A \in ENV(X, A)$ such that (C, η_C) imitates (A, η_A) with respect to *OBS* and for all identifiers $\mathbf{x} \in X$, $\eta_A(\mathbf{x})$ has the same type as $\eta_C(\mathbf{x})$.

We say that an algebra is a universal model of a specification *SPEC* with respect to some set of observations if it is a universal model of the set of *SPEC*-algebras with respect to that set of observations.

For example, a maximally nondeterministic Mob-algebra is a universal model of the specification Mob with respect to the set of observations that contains the single *NOAL* program `next(x)`, where `x` has nominal type Mob. To see this, suppose A is a maximally nondeterministic Mob-algebra whose carrier set for Mob is $\bot$ plus all finite sets of integers. Let C be a Mob-algebra. Let $\eta_C \in ENV(\{\mathtt{x} : \mathtt{Mob}\}, C)$ be such that the set of integers waiting in $\eta_C(\mathtt{x})$ is some set R. (That is, for all integers i, $i \in R$ if and only if $\mathcal{M}[\![\mathtt{waiting?(x, i)}]\!](C, \eta_C[i/\mathtt{i}]) = \{true\}$.) Let $\eta_A \in ENV(\{\mathtt{x} : \mathtt{Mob}\}, A)$ be such that $\eta_A(\mathtt{x}) = R$. Then by the specification of Mob,

$$\mathcal{M}[\![\mathtt{next(x)}]\!](C, \eta_C) \subseteq R = \mathcal{M}[\![\mathtt{next(x)}]\!](A, \eta_A). \tag{6.22}$$

So (C, η_C) imitates (A, η_A) with respect to this set of observations and thus A is universal.

Unfortunately, not every specification has a universal model. For example, the specification of the type Crowd given in Figure 5.1 has no universal model with respect to type-safe NOAL programs. The reason is that different Crowd-algebras may have different algorithms for their `next` operation, and since the `next` operation must be deterministic, there is no single Crowd-algebra that captures all this behavior. Notice that the abstract values of the type Crowd do not fully describe the behavior of Crowd objects, since from the abstract value one cannot tell what waiting integer will be chosen by `next`, unlike the abstract values of type PSchd.

When a specification has a universal model, however, it eases the problem of finding subtype relations, as the following lemma shows.

Lemma 6.3.7. Let Σ be a signature and let *SPEC* be a nonempty set of Σ-algebras. Let *OBS* be a set of observations. Let $\leq$ be a binary relation on type symbols.

Suppose A is a universal model of *SPEC* with respect to *OBS*. Then $\leq$ is a subtype relation on the types of *SPEC* with respect to *OBS* if and only if for all sets of typed identifiers X and for all environments $\eta_1 \in ENV(X, A)$ such that η_1 obeys $\leq$, there is some nominal environment $\eta_2 \in ENV(X, A)$ such that (A, η_1) imitates (A, η_2) with respect to *OBS*.

Proof: Let A be a universal model of *SPEC*. Suppose that for all sets of typed identifiers X and for all environments $\eta_1 \in ENV(X, A)$ such that η_1 obeys $\leq$, there is some nominal environment $\eta_2 \in ENV(X, A)$ such that (A, η_1) imitates (A, η_2) with respect to *OBS*. Let $C \in SPEC$ be an algebra and let $\eta_C \in ENV(X, C)$ be an environment that obeys $\leq$.

Since A is a universal model of *SPEC*, there is some environment $\eta_1 \in ENV(X, A)$ that obeys $\leq$ and such that (C, η_C) imitates (A, η_1) with respect to *OBS*. Since η_1 obeys $\leq$, by hypothesis there is a nominal environment $\eta_2 \in ENV(X, A)$ such that (A, η_1) imitates (A, η_2) with respect to *OBS*. By Lemma 4.2.2, the imitates relation with respect to *OBS* is transitive. Therefore (C, η_C) imitates (A, η_2) with respect to *OBS*. So by definition $\leq$ is a subtype relation.

Conversely, suppose that $\leq$ is a subtype relation on the types of *SPEC* with respect to *OBS*. Let X be a set of typed identifiers. Let $\eta_1 \in ENV(X, A)$ be an environment that obeys $\leq$. Since $\leq$ is a subtype relation, there is some $B \in SPEC$ and some nominal environment $\eta_B \in ENV(X, B)$ such that (A, η_1) imitates (B, η_B) with respect to *OBS*. Since A is universal, there is some nominal environment $\eta_2 \in ENV(X, A)$ such that (B, η_B) imitates (A, η_2) with respect to *OBS*. Since the imitates relation with respect to *OBS* is transitive, (A, η_1) imitates (A, η_2) with respect to *OBS*. ∎

As a corollary to the above lemma, if we have a universal model of a specification, we only need to find a simulation relation between the universal model and itself that witnesses a binary relation $\leq$ to prove that $\leq$ is a subtype relation.

Corollary 6.3.8. Let *SPEC* be a set of Σ-algebras. Let *NomSig* be a nominal signature map. Let $\leq$ be a binary relation on type symbols.

If A is a universal model of SPEC and if there is a simulation relation $\mathcal{R}$ between A and itself for NomSig and $\leq$ such that $\mathcal{R}$ witnesses $\leq$, then $\leq$ is a subtype relation on the types of SPEC with respect to the set of type-safe NOAL programs over NomSig and $\leq$.

Proof: Suppose that A is a universal model of *SPEC* and that there is a simulation relation $\mathcal{R}$ between A and itself that witnesses $\leq$. Let $\eta_1 \in ENV(X, A)$ be an environment that obeys $\leq$. Since $\mathcal{R}$ witnesses $\leq$, one can build a nominal environment

$\eta_2 \in ENV(X, A)$ such that $\eta_1 \mathcal{R} \eta_2$. By Theorem 6.2.3, (A, η_1) imitates (A, η_2). Since A is a universal model of *SPEC*, it follows by Lemma 6.3.7 that $\leq$ is a subtype relation. ∎

As an example of proving that a binary relation is a subtype relation on the types of a specification with a universal model, consider the specification IPT, with types `IntPair` and `IntTriple`. Let *NomSig* be the nominal signature map of IPT and let $\leq$ be smallest binary relation on the types of IPT such that `IntTriple` $\leq$ `IntPair`. The algebra A given in Figure 2.5 is a universal model of IPT. Let $\mathcal{R}$ be the smallest typed family of relations such that for all types T, if $q \in A_{\mathtt{T}}$, then $q \; \mathcal{R}_{\mathtt{T}} \; q$, and for all $\langle q_1, q_2, q_3 \rangle, \langle q_1, q_2, q_4 \rangle \in A_{\mathtt{IntTriple}}$ and $\langle q_1, q_2 \rangle \in A_{\mathtt{IntPair}}$,

$$\langle q_1, q_2, q_3 \rangle \;\; \mathcal{R}_{\mathtt{IntPair}} \;\; \langle q_1, q_2 \rangle \tag{6.23}$$

$$\langle q_1, q_2 \rangle \;\; \mathcal{R}_{\mathtt{IntPair}} \;\; \langle q_1, q_2, q_3 \rangle \tag{6.24}$$

$$\langle q_1, q_2, q_3 \rangle \;\; \mathcal{R}_{\mathtt{IntPair}} \;\; \langle q_1, q_2, q_4 \rangle. \tag{6.25}$$

We showed earlier in this chapter that $\mathcal{R}$ is a simulation relation. We now note that $\mathcal{R}$ witnesses $\leq$ because every proper instance of `IntTriple` is related to an instance of `IntPair` with the same first and second components. Since A is a universal model of IPT, it follows by Corollary 6.3.8 that $\leq$ is a subtype relation on the types of IPT with respect to the set of all type-safe NOAL programs over *NomSig* and $\leq$.

6.4 Examples Proofs of Subtype Relations

In the following subsections we prove several examples of subtype relations with respect to type-safe NOAL programs. All the examples were introduced earlier chapters, where much smaller sets of observations were used.

6.4.1 Schedulers

In Chapter 5 we discussed the specification MCP, with three types of schedulers: `Mob`, `Crowd`, and `PSchd` (see Figure 2.6 on page 39, Figure 5.1 on page 81, and Figure 4.2 on page 63). The specification MCP is interesting because it involves a nondeterministic type, `Mob`, and two incompletely specified types: `Mob` and `Crowd`. Furthermore, MCP does not have a universal model.

Let $\leq$ be the smallest reflexive relation on the types of MCP such that $\texttt{Crowd} \leq \texttt{Mob}$ and $\texttt{PSchd} \leq \texttt{Mob}$. (The relation $\leq$ is depicted in Figure 5.2 on page 82.) In Chapter 5 we argued that $\leq$ is a subtype relation on the types of MCP with respect to a set of three NOAL programs. The following lemma shows that $\leq$ is a subtype relation with respect to all type-safe NOAL programs over the nominal signature map of MCP and $\leq$.

Lemma 6.4.1. The relation $\leq$ described above is a subtype relation with respect to all type-safe NOAL programs over the nominal signature map of MCP and $\leq$.

Proof: Let C be a MCP-algebra. Although there is no universal model of MCP, there is an algebra, call it A, with the same carrier set, abstract functions, and operations as C, except that the $A_{\texttt{next}_{\texttt{Mob}\to\texttt{Int}}}$ operation is maximally nondeterministic. This algebra A is a MCP-algebra, because the specification of `Mob` permits the $\texttt{next}_{\texttt{Mob}\to\texttt{Int}}$ operation to be nondeterministic.

Let us say that an integer i is *waiting* in a scheduler if the result of the generic invocation `waiting?(x,i)` in an environment where `x` denotes the scheduler and `i` denotes the integer i is *true*. Then we can define a homomorphic relation $\mathcal{R}$ between C and A for the nominal signature map of MCP and $\leq$ as follows. Let $\mathcal{R}$ be the smallest typed family of relations between C and A, such that for all types $\texttt{T}$, $\mathcal{R}_{\texttt{T}}$ is the identity on $C_{\texttt{T}}$ (which is the same as $A_{\texttt{T}}$), every proper instance of `Crowd` or `PSchd` is related to an instance of `Mob` with the same set of waiting integers by $\mathcal{R}_{\texttt{Mob}}$ and in addition:

$$\mathcal{R}_{\texttt{Crowd}} \subseteq \mathcal{R}_{\texttt{Mob}} \tag{6.26}$$

$$\mathcal{R}_{\texttt{PSchd}} \subseteq \mathcal{R}_{\texttt{Mob}}. \tag{6.27}$$

By this construction relationships at types `Crcwd` and `PSchd` hold at type `Mob` and $\mathcal{R}$ witnesses $\leq$.

We now show that $\mathcal{R}$ is a homomorphic relation. Let X and environments $\eta_1 \in ENV(X, C)$ and $\eta_2 \in ENV(X, A)$ be such that $\eta_1 \, \mathcal{R} \, \eta_2$. Suppose $\texttt{g}(\vec{\texttt{x}})$ is a generic invocation of some nominal type $\texttt{T}$. If none of thc identifiers in $\vec{\texttt{x}}$ have nominal type `Mob`, then by construction $\eta_1(\vec{\texttt{x}}) = \eta_2(\vec{\texttt{x}})$ and thus $\mathcal{M}[\![\texttt{g}(\vec{\texttt{x}})]\!](C, \eta_1) = \mathcal{M}[\![\texttt{g}(\vec{\texttt{x}})]\!](A, \eta_2)$. Since each $\mathcal{R}_{\texttt{T}}$ contains the identity relation on $A_{\texttt{T}}$, it follows that $\mathcal{M}[\![\texttt{g}(\vec{\texttt{x}})]\!](C, \eta_1) \, \mathcal{R}_{\texttt{T}} \, \mathcal{M}[\![\texttt{g}(\vec{\texttt{x}})]\!](A, \eta_2)$. So it remains to deal with generic invocations whose arguments have nominal type `Mob`. In the following cases, suppose that $\texttt{x} : \texttt{Mob} \in X$ and $\texttt{i} : \texttt{Int} \in X$.

- Suppose the generic invocation is `ins(x,i)`. Let q be the only possible result of $\mathcal{M}[\![\texttt{ins(x,i)}]\!](C,\eta_1)$ and let r be the only possible result of $\mathcal{M}[\![\texttt{ins(x,i)}]\!](A,\eta_2)$. Since $\eta_1(\texttt{x})\,\mathcal{R}_{\texttt{Mob}}\,\eta_2(\texttt{x})$, both $\eta_1(\texttt{x})$ and $\eta_2(\texttt{x})$ have the same set of waiting integers, call it s. By the specification of `ins` in the three scheduler types, the set of waiting integers for q is $s \cup \{i\}$, where i is $\eta_1(\texttt{i})$. Since $\eta_1(\texttt{i})\,\mathcal{R}_{\texttt{Int}}\,\eta_2(\texttt{i})$ and $\mathcal{R}_{\texttt{Int}}$ is the identity, the set of waiting integers for r is also $s \cup \{i\}$. So by definition of $\mathcal{R}_{\texttt{Mob}}$, $q\,\mathcal{R}_{\texttt{Mob}}\,r$.

- Suppose the generic invocation is `waiting?(x,i)`. Since $\eta_1(\texttt{x})\,\mathcal{R}_{\texttt{Mob}}\,\eta_2(\texttt{x})$, both $\eta_1(\texttt{x})$ and $\eta_2(\texttt{x})$ have the same set of waiting integers. As above, the denotation of `i` must be the same in both environments. So by definition of "waiting integer," the denotation of `waiting?(x,i)` must be the same in both environments. Since $\mathcal{R}_{\texttt{Bool}}$ is the identity on the carrier set of `Bool`, we have

$$\mathcal{M}[\![\texttt{waiting?(x,i)}]\!](C,\eta_1)\,\mathcal{R}_{\texttt{Bool}}\,\mathcal{M}[\![\texttt{waiting?(x,i)}]\!](A,\eta_2).$$

- Suppose the generic invocation is `empty?(x)`. As above, both $\eta_1(\texttt{x})$ and $\eta_2(\texttt{x})$ have the same set of waiting integers. So by the specification of the `empty?` operations of the various types, the denotation of `empty?(x)` is the same in both environments.

- Suppose the generic invocation is `next(x)`. Since $\eta_1(\texttt{x})\,\mathcal{R}_{\texttt{Mob}}\,\eta_2(\texttt{x})$, both $\eta_1(\texttt{x})$ and $\eta_2(\texttt{x})$ have the same set of waiting integers, call it s. By definition of $\mathcal{R}$, either $\eta_2(\texttt{x})$ denotes an instance of `Mob`, or $\eta_1(\texttt{x}) = \eta_2(\texttt{x})$ if $\eta_2(\texttt{x})$ denotes an instance of `PSchd` or `Crowd`. In the latter case, the result is trivial, so suppose $\eta_2(\texttt{x})$ denotes an instance of `Mob`. Recall that the algebra A is defined so that

$$\mathcal{M}[\![\texttt{next(x)}]\!](A,\eta_2) = s \tag{6.28}$$

because the operation $A_{\texttt{next}_{\texttt{Mob}\to\texttt{Int}}}$ is maximally nondeterministic. By the specifications of the three types, the set of possible results for $\mathcal{M}[\![\texttt{next(x)}]\!](C,\eta_1)$ must be a subset of s. Therefore,

$$\mathcal{M}[\![\texttt{next(x)}]\!](C,\eta_1) \subseteq \mathcal{M}[\![\texttt{next(x)}]\!](A,\eta_2). \tag{6.29}$$

Since $\mathcal{R}_{\texttt{Int}}$ is the identity relation on the carrier set of `Int`, it follows that $\mathcal{M}[\![\texttt{next(x)}]\!](C,\eta_1)\,\mathcal{R}_{\texttt{Int}}\,\mathcal{M}[\![\texttt{next(x)}]\!](A,\eta_2)$, since this means that for each $q \in$

$\mathcal{M}[\![\texttt{next(x)}]\!](C,\eta_1)$, there is some $r \in \mathcal{M}[\![\texttt{next(x)}]\!](A,\eta_2)$ such that $q = r$, or in other words that $\mathcal{M}[\![\texttt{next(x)}]\!](C,\eta_1) \subseteq \mathcal{M}[\![\texttt{next(x)}]\!](A,\eta_2)$.

So this $\mathcal{R}$ is a homomorphic relation. By construction $\mathcal{R}$ is also V-identical and bistrict; hence $\mathcal{R}$ is a simulation relation. Since $\mathcal{R}$ witnesses $\leq$, the result follows by Theorem 6.3.2. ∎

6.4.2 OneOfs

In this subsection we will show how the general subtyping rule given by Cardelli for his immutable variant types [Car84] holds for our `OneOf` types. This example shows that our definition of subtype relations encompasses Cardelli's rule for `OneOf` types. That is, a `OneOf` type with fewer tags is a subtype of a `OneOf` type with more tags, provided each type associated with a tag in the first `OneOf` type is a subtype of the corresponding type in the second `OneOf` type.

We discussed `OneOf` types in Chapters 2 and 4, where they are used to model exceptions. We now introduce a slightly more general specification of `OneOf` types, which properly treats tags whose associated types may have supertypes. A representative `OneOf` type specification is given in Figure 6.2, where the type is abbreviated to O12. In a `OneOf` type name, the order of the tag declarations (`ni:Si`) does not matter. The trait that specifies the carrier set and abstract functions of `OneOf` types is given in Figure 2.7 on page 40. Note that the `value[`T`]` operations are defined for all types T; their specification uses the presumed subtype relation $\leq$ that we will define below in this example.

Let S be a specification. The specification S describes the data found in the simplest OneOf types and is a basis for the inductive construction that follows. Let $\leq_S$ be a reflexive subtype relation on the types of S with respect to type-safe NOAL programs over the nominal signature map of S and $\leq_S$. We assume that, for every S-algebra C, there is some S-algebra A and a simulation relation between C and A for the nominal signature map of S and $\leq_S$ such that the simulation relation witnesses $\leq_S$.

Let O be a specification that includes S and several `OneOf` types of the form `OneOf[n1:S1,...,nk:Sk]`, where the `Si` are either types from the specification S or other `OneOf` types of this form. (So that all the algebras that satisfy the specification O are flat, as required for NOAL programs in Appendix C, we require that if an `Si` is a type

Figure 6.2: Specification of the type `O12` = `OneOf`$[n_1 : S_1, n_2 : S_2]$.

```
O12 immutable type
    class ops [make_n₁, make_n₂]
        instance ops [hasTag?, value[T]]
    based on sort OneOf from OneOf[n₁ : S₁, n₂ : S₂]

    op make_n₁(c:O12Class, x: S₁) returns(o:O12)
        effect o = make_n₁(x)

    op make_n₂(c:O12Class, x: S₂) returns(o:O12)
        effect o = make_n₂(x)

    op hasTag?(o:012, t: Tag) returns(b:Bool)
        effect b = hasTag?(o, t)

    op value[T](o:O12, t: Tag) returns(r:T)
        requires hasTag?(o, t)
            & ((t = n₁) ⇒ S₁ ≤ T)
            & ((t = n₂) ⇒ S₂ ≤ T)
        effect ((t = n₁) ⇒ r = val_n₁(o))
            & ((t = n₂) ⇒ r = val_n₂(o))
```

from the specification S, then that type is a type whose carrier set in all S-algebras is flat. For example, we would disallow `OneOf[b:BoolStream,i:IntStream]` as a type of `O`, because the carrier sets of `BoolStream` and `IntStream` are not flat.) Let *NomSig* be the nominal signature map of O.

We inductively define a presumed subtype relation $\leq$ on the types of O as follows. As the basis, for all types `S1` and `S2` from the specification S, $\texttt{S1} \leq \texttt{S2}$ if and only if $\texttt{S1} \leq_S \texttt{S2}$. Furthermore, $\texttt{Tag} \leq \texttt{Tag}$, $\texttt{TagClass} \leq \texttt{TagClass}$, and for all class types `OClass` introduced by O, $\texttt{OClass} \leq \texttt{OClass}$. For the `OneOf` types introduced by the specification O,

$$\begin{array}{l}\texttt{OneOf}[\texttt{n}_1 : \texttt{S}_1, \ldots, \texttt{n}_j : \texttt{S}_j] \leq \texttt{OneOf}[\texttt{n}_1 : \texttt{T}_1, \ldots, \texttt{n}_j : \texttt{T}_j, \ldots, \texttt{n}_k : \texttt{T}_k] \\ \Leftrightarrow \forall(\texttt{i} \in \{1, \ldots, \texttt{j}\})\texttt{S}_\texttt{i} \leq \texttt{T}_\texttt{i}.\end{array} \tag{6.30}$$

That is, for `OneOf` types, one type is related by $\leq$ to another only if the first has a subset of the second's tags and the corresponding types are related by $\leq$. Note that we do not allow `OneOf` types to be recursively defined.

The next lemma says that $\leq$ is a subtype relation with respect to the set of type-safe NOAL programs over *NomSig* and $\leq$. In the proof we inductively build a simulation relation with the properties required to show that $\leq$ is a subtype relation.

Lemma 6.4.2. The relation $\leq$ described above is a subtype relation on the types of O with respect to the set of type-safe NOAL programs over *NomSig* and $\leq$.

Proof: Let C be an O-algebra. Let $C_{(S)}$ be the $SIG(S)$-reduct of C. Since $\leq_S$ is a subtype relation, there is some S-algebra $A_{(S)}$ such that every environment over $C_{(S)}$ that obeys $\leq_S$ imitates a nominal environment over $A_{(S)}$. Since the specification of a `OneOf` type of the form `OneOf[n1:S1,...,nk:Sk]` does not constrain the types `Si`, there is some O-algebra A whose $SIG(S)$-reduct is $A_{(S)}$. Without loss of generality, we can assume that each `value[`T`]` operation of A is maximally nondeterministic when its **requires** clause is not satisfied, because this behavior is permitted by our specification of `OneOf` types.

We define a homomorphic relation between C and A for *NomSig* and $\leq$ as follows.

First some terminology. We say that a `OneOf` object o has tag `t` if the result of $\mathcal{M}[\![\texttt{hasTag?(o,t)}]\!](C, \eta_o)$ is *true*, where $\eta_o(\texttt{o}) = o$. We say that `value[T]`$(o,$`t`$)$ is q if the only possible result of $\mathcal{M}[\![\texttt{value[T](o,t)}]\!](C, \eta_o)$ is q, where $\eta_o(\texttt{o}) = o$.

By our assumption above, there is a simulation relation $\mathcal{R}^S$ between $C_{(\mathtt{S})}$ and $A_{(\mathtt{S})}$ that witnesses $\leq_S$. We define a homomorphic relation $\mathcal{R}$ between C and A by structural induction on type expressions to be the smallest typed family of relations between C and A, such that:

- $\mathcal{R}^S \subseteq \mathcal{R}$; that is, if $\mathtt{T}$ is a type of the specification S, then $\mathcal{R}_{\mathtt{T}}$ contains $\mathcal{R}^S_{\mathtt{T}}$.

- Let $O_1 = \mathtt{OneOf}[\mathtt{n_1} : \mathtt{S1}, \ldots, \mathtt{n_j} : \mathtt{S_j}]$ be a `OneOf` type introduced by O. Then $\mathcal{R}_{O_1}$ is such that $q\ \mathcal{R}_{O_1}\ r$ if q and r are both $\perp$ or q and r are proper instances of some `OneOf` type introduced by O such that q and r have the same tag, which is some $\mathtt{n_i}$ in the set of tags of O_1 and

$$\begin{aligned} \mathtt{value}[\mathtt{S_i}](q, \mathtt{n_i}) &= q' && (6.31)\\ \mathtt{value}[\mathtt{S_i}](r, \mathtt{n_i}) &= r' && (6.32)\\ q' \ \mathcal{R}_{\mathtt{S_i}} &\ r'. && (6.33)\end{aligned}$$

- For all class types `OClass` introduced by O, $\mathcal{R}_{\mathtt{OClass}}$ is such that $\perp$ in C is related to $\perp$ in A and the only proper object of type `OClass` in C is related to the proper object of type `OClass` in A.

- We have not specified the type **Tag** in detail. But let us agree that it is generated by class operations of the form `t(Tag)`, which we abbreviate to `t` for all "tag names" `t`. Furthermore, two tag objects are the same only if they are the result of the same operation `t(Tag)`. We define $\mathcal{R}_{\mathtt{Tag}}$ as the smallest relation between the carrier sets of **Tag** in C and A such that for all tag names `t`, $\mathcal{M}[\![\mathtt{t(Tag)}]\!](C, \emptyset)\ \mathcal{R}_{\mathtt{Tag}}\ \mathcal{M}[\![\mathtt{t(Tag)}]\!](A, \emptyset)$. That is, $\mathcal{R}_{\mathtt{Tag}}$ only relates one tag object to another if both are the "same tag."

Before showing that $\mathcal{R}$ is a homomorphic relation, we first need to show that $\mathcal{R}$ witnesses $\leq$ and satisfies Formula 6.1. This is proved by a (double) structural induction on type expressions. That is, we show for all types `T` and for all types `S` such that $\mathtt{S} \leq \mathtt{T}$, that $\mathcal{R}$ witnesses $\leq$ and that Formula 6.1 holds. As a basis, the claim holds for all types introduced by the specification S, since $\mathcal{R}^S$ satisfies the claim by hypothesis. The type **Tag** is only related to itself by $\leq$ and by construction $C_{\texttt{Tag}} \mathcal{R}_{\texttt{Tag}} A_{\texttt{Tag}}$. Furthermore, a class type introduced by O is related only to itself by $\leq$, and by construction $C_{\texttt{OClass}} \mathcal{R}_{\texttt{OClass}} A_{\texttt{OClass}}$.

For the inductive step, let O_1 and O_2 be `OneOf` types introduced by the specification O such that $O_1 \leq O_2$ and assume for all their component types S and T Formulas 6.1 and 6.16 hold. Since $O_1 \leq O_2$, O_1 must have the form $\texttt{OneOf}[\texttt{n}_1 : \texttt{S}_1, \ldots, \texttt{n}_\texttt{j} : \texttt{S}_\texttt{j}]$ and O_2 m ust have the form $\texttt{OneOf}[\texttt{n}_1 : \texttt{T}_1, \ldots, \texttt{n}_\texttt{j} : \texttt{T}_\texttt{j}, \ldots, \texttt{n}_\texttt{k} : \texttt{T}_\texttt{k}]$ where for i from 1 to j, $\texttt{S}_\texttt{i} \leq \texttt{T}_\texttt{i}$. The inductive assumption is that the formulas hold for each pair of types $\texttt{S}_\texttt{i} \leq \texttt{T}_\texttt{i}$ where $i \in \{1, \ldots, j\}$. Then $C_{O_1} \mathcal{R}_{O_2} A_{O_2}$, because O_1 has a subset of the tags of O_2, and so every proper element of C_{O_1} has the tag of some proper element of A_{O_2}, and because its value at that tag is related to some element in A by $\mathcal{R}_{\texttt{S}_\texttt{i}}$, where $\texttt{S}_\texttt{i}$ is the type associated with the tag. Suppose $q \mathcal{R}_{O_1} r$. Then by construction, q and r have the same tag, say $\texttt{n}_\texttt{i}$, and their values at that tag are related by $\mathcal{R}_{\texttt{S}_\texttt{i}}$. By the inductive hypothesis, $\mathcal{R}_{\texttt{S}_\texttt{i}} \subseteq \mathcal{R}_{\texttt{T}_\texttt{i}}$, so their values at that tag are related by $\mathcal{R}_{\texttt{T}_\texttt{i}}$. So $q \mathcal{R}_{O_1} r$ by construction and therefore $\mathcal{R}_{O_1} \subseteq \mathcal{R}_{O_2}$.

We now show that $\mathcal{R}$ is a homomorphic relation. Let X and environments $\eta_1 \in ENV(X, C)$ and $\eta_2 \in ENV(X, A)$ be such that $\eta_1 \mathcal{R} \eta_2$. Suppose $\mathbf{g}(\vec{\mathbf{x}})$ is a generic invocation of some nominal type T. If T is not a `OneOf` type and none of the identifiers in $\vec{\mathbf{x}}$ has a `OneOf` type, then for each $\mathbf{x}_i : \mathrm{T}_i$, $\eta_1(\mathbf{x}_i) \mathcal{R}^S_{\mathrm{T}_i} \eta_2(\mathbf{x}_i)$, because $\mathcal{R}$ does not relate proper instances of `OneOf` types at the types of S and because $\eta_1 \mathcal{R} \eta_2$. Since $\mathcal{R}^S$ is a homomorphic relation, we have $\mathcal{M}[\![\mathbf{g}(\vec{\mathbf{x}})]\!](C, \eta_1) \mathcal{R}_\mathrm{T} \mathcal{M}[\![\mathbf{g}(\vec{\mathbf{x}})]\!](A, \eta_2)$. So it remains to deal with generic invocations involving the `OneOf` types. There are several cases.

- The generic invocation is `make_n(x,y)` for some tag name n and some identifiers $\texttt{x} : \texttt{OneOf}[\ldots, \texttt{n} : \texttt{T}, \ldots]\texttt{Class}$ and $\texttt{y} : \texttt{S}$ where $\texttt{S} \leq \texttt{T}$. This generic invocation has nominal type `OneOf[..., n:T, ...]`. Since $\eta_1 \mathcal{R} \eta_2$, $\eta_1(\texttt{y}) \mathcal{R}_\texttt{S} \eta_2(\texttt{y})$, and thus $\eta_1(\texttt{y}) \mathcal{R}_\texttt{T} \eta_2(\texttt{y})$, because $(\texttt{S} \leq \texttt{T}) \Rightarrow \mathcal{R}_\texttt{S} \subseteq \mathcal{R}_\texttt{T}$. So by definition of $\mathcal{R}$ at `OneOf` types,

$$\mathcal{M}[\![\texttt{make\_n}(\texttt{x},\texttt{y})]\!](C, \eta_1)\, \mathcal{R}_{\texttt{OneOf}[\ldots,\texttt{n}:\texttt{T},\ldots]}\, \mathcal{M}[\![\texttt{make\_n}(\texttt{x},\texttt{y})]\!](A, \eta_2). \qquad (6.34)$$

- The generic invocation is `hasTag?(x,y)` for some `OneOf` type O_1 and some identifiers $\texttt{x} : O_1$ and $\texttt{y} : \texttt{Tag}$. This generic invocation has nominal type `Bool`. Since $\eta_1 \mathcal{R} \eta_2$, $\eta_1(\texttt{y})$ and $\eta_2(\texttt{y})$ must denote the same tag name. Furthermore, $\mathcal{R}$ only relates objects at a `OneOf` type if they have the same tag. So

$\mathcal{M}[\![\texttt{hasTag?(x,y)}]\!](C,\eta_1) = \mathcal{M}[\![\texttt{hasTag?(x,y)}]\!](A,\eta_2)$. Since $\mathcal{R}_{\texttt{Bool}}$ is the identity, we have

$$\mathcal{M}[\![\texttt{hasTag?(x,y)}]\!](C,\eta_1)\,\mathcal{R}_{\texttt{Bool}}\,\mathcal{M}[\![\texttt{hasTag?(x,y)}]\!](A,\eta_2). \qquad (6.35)$$

- The generic invocation is `value[T](x,y)` for some type `T`, some `OneOf` type O_1, and some identifiers $\texttt{x} : O_1$ and $\texttt{y} : \texttt{Tag}$. This generic invocation has nominal type `T`. Since $\eta_1 \mathcal{R} \eta_2$, $\eta_1(\texttt{y})$ and $\eta_2(\texttt{y})$ denote the same tag name. Therefore, the requires clause for the appropriate `value[T]` operations is satisfied in C if and only if it is satisfied in A. We chose A so that the `value[T]` operation is maximally nondeterministic when its requires clause is not satisfied; therefore, if the requires clause is not satisfied, every element of the result in C is related to some element of the result in A by $\mathcal{R}_{\texttt{T}}$, because $\leq$ is reflexive and $C_{\texttt{T}} \mathcal{R}_{\texttt{T}} A_{\texttt{T}}$. So suppose the requires clause is satisfied; that is, suppose that both $\eta_1(\texttt{x})$ and $\eta_2(\texttt{x})$ have tag $\eta_2(\texttt{y})$. By definition of $\mathcal{R}_{O_1}$ it must be that

$$\mathcal{M}[\![\texttt{value[T](x,y)}]\!](C,\eta_1)\,\mathcal{R}_{\texttt{T}}\,\mathcal{M}[\![\texttt{value[T](x,y)}]\!](A,\eta_2) \qquad (6.36)$$

So $\mathcal{R}$ is a homomorphic relation.

The relation $\mathcal{R}$ is also a simulation relation. It is V-identical, because $\mathcal{R}^S$ is V-identical and $\mathcal{R}$ does not add new relationships between proper elements of the visible types. It is also bistrict and satisfies Formula 6.1 by construction.

Since $\mathcal{R}$ is a simulation relation that witnesses $\leq$, by Theorem 6.3.2, it follows that $\leq$ is a subtype relation with respect to type-safe NOAL programs over *NomSig* and $\leq$. ∎

6.4.3 Exceptions

As discussed in Chapter 5, a subtype may have fewer exceptions than its supertypes. In this section we show that this rule also holds with respect to all type-safe NOAL programs.

Consider again the specification M4, which consists of the types `Mob` and `Mob3` as specified in Figure 2.6 on page 39 and Figure 5.4 on page 95. The type `Mob3` is like `Mob`, except that it can also signal an exception "empty(nil)." It does this by returning the denotation of `make_empty(NE,nil(Null))` instead of `make_normal(NE,`i`)`, for some integer i (where NE abbreviates OneOf[normal: Int, empty: Null]). Since we are concerned with exceptions, we again consider that the return type of the **next** operation of type Mob is OneOf[normal: Int]. Let *NomSig* be the nominal signature map of M4.

Let $\leq$ be the smallest reflexive relation on the types of M4 such that Mob $\leq$ Mob3 and

$$\texttt{OneOf[normal : Int]} \leq \texttt{OneOf[normal : Int, empty : Null]}.$$

Unlike the previous section, the types Mob3 and Mob are specified using the OneOf types. Therefore we cannot separate the proof of a subtype relationship for M4 into a proof for non-OneOf types and a proof for OneOf types, as in the previous section.

Let C be an M4-algebra. Let A be an M4-algebra with the same carrier sets and abstract functions as C but with a maximally nondeterministic $\texttt{next}_{\texttt{Mob3}\to\texttt{Int}}$ and value[T] operations. Such an M4-algebra exists, because of the way the types are specified.

We define a simulation relation $\mathcal{R}$ between C and A for *NomSig* and $\leq$ as follows. Let $\mathcal{R}$ be the smallest typed family of relations between C and A such that

- for each type T, $\mathcal{R}_{\texttt{T}}$ contains the identity on $C_{\texttt{T}}$ (which is the same as $A_{\texttt{T}}$),
- $\mathcal{R}_{\texttt{Mob3}}$ relates all proper instances of Mob to instances of either Mob3 or Mob with the same elements (waiting integers), and
- $\mathcal{R}_{\texttt{OneOf[normal:Int,empty:Null]}}$ relates each instance q of OneOf[normal: Int] such that value[Int](q, normal) is some integer i both to itself and also to all instances r of OneOf[normal: Int, empty: Null] with the same tag such that value[Int](r, normal) is i.

It is straightforward to show that $\mathcal{R}$ is a homomorphic relation. Clearly $\mathcal{R}$ is also V-identical, bistrict and satisfies Formula 6.1. So $\mathcal{R}$ is a simulation relation.

By construction $\mathcal{R}$ witnesses $\leq$ and so $\leq$ is a subtype relation on the types of M4 with respect to all type-safe NOAL programs over *NomSig* and $\leq$.

Chapter 7

Hoare-style Verification for NOAL Programs

In this chapter we present a Hoare logic for the verification of NOAL programs. The key idea is using simulation relations to translate assertions tailored to a subtype into assertions tailored to a supertype. We also draw some conclusions about how a type system aids verification and discuss the role that observable assertions play in verification.

Our verification technique has the property that if a subtype is not explicitly used in a NOAL function definition, then it is ignored during verification of that function. Because of this property, if one verifies a NOAL function and then later adds new subtypes to some of the function's nominal argument types, then the verification does not have to be repeated.

As usual, the specifications of each type's operations and the specification of each recursively-defined NOAL function is an axiom. The axiom used for a particular generic invocation is determined by the nominal type of the generic invocation's first argument (that is, using static instead of dynamic overloading). When the program explicitly passes an expression of nominal type S as an argument when the nominal type of the corresponding formal argument is T, simulation relations are used to translate knowledge about the actual expression from type S to type T.

A verification technique for NOAL programs is *sound* if whenever one concludes by using that technique that a program satisfies its specification with respect to the set of all type-safe NOAL programs, then that program does indeed satisfy its specification. After presenting our Hoare logic, we show that it is sound.

7.1 A Hoare Logic for NOAL

In this section we give a Hoare logic [Hoa69] for NOAL programs. We take a total correctness approach, since our specifications require termination whenever the precondition is met. Our logic is sound, but it is not complete, since we are unable to reason about nontermination as would be required to deal with NOAL's lazy evaluation and angelic choice expressions in a complete way.

Although NOAL is applicative, we use a Hoare logic because NOAL is nondeterministic and because we are ultimately interested in verification of imperative programs, for which Hoare-style reasoning is an accepted technique.

The fundamental formulas of a Hoare logic are called Hoare-triples. Hoare-triples are written $P \; \{\mathrm{v} \leftarrow E\} \; Q$ and consist of a *precondition* P, a *result identifier* v, an expression E, and a *postcondition* Q. (The name of the result identifier can be chosen at will.) In a Hoare logic for an imperative language, the precondition describes the state before the execution of a statement, and the postcondition describes the changed state that results from the statement's execution. In NOAL, however, expressions have results but do not change the environment in which they execute. So in our logic, the precondition of a Hoare-triple describes the environment, and the postcondition describes the environment that results from binding the possible results of the expression to the result identifier (v). So that the notation does not cause confusion, the result identifier in a Hoare-triple cannot occur free in the precondition. Otherwise one might think that the execution of E changes the binding of the result identifier in the surrounding environment, whereas we are only using suggestive notation that shows what identifier will be used to denote the possible results of E in the postcondition.

Definition 7.1.1 (Hoare-triple for *SPEC*). Let *SPEC* be a specification. Then $P \; \{\mathrm{y} \leftarrow E\} \; Q$ is a *Hoare-triple for SPEC* if and only if E is a type-safe NOAL expression P and Q are *SPEC*-assertions, y has the same nominal type as E, and y does not occur free in P.

We do not mention the specification name when it is clear from context.

Intuitively, $P \; \{\mathrm{v} \leftarrow E\} \; Q$ is true if whenever P holds, then the execution of E terminates, and all possible results satisfy Q. The semantics of a Hoare-triple is given by the following definition, which is similar to the definition of satisfies for function specifications.

Definition 7.1.2 (models for Hoare-triples). Let *SPEC* be a specification. Let $P\ \{\mathrm{v} \leftarrow E\}\ Q$ be a Hoare-triple for *SPEC*. Let X be a set of free identifiers that contains all the free identifiers of P, E, and Q except v. Let A be a *SPEC*-algebra, and $\eta \in ENV(X, A)$ an environment that is proper and obeys the presumed subtype relation of *SPEC*. Let η' be the environment that extends η by binding each free function identifier in E to the denotation in A of the corresponding recursive function definition. We say that (A, η) *models* $P\ \{\mathrm{v} \leftarrow E\}\ Q$ and write

$$(A, \eta) \models P\ \{\mathrm{v} \leftarrow E\}\ Q$$

if and only if whenever $(A, \eta) \models P$, then for all possible results $q \in \mathcal{M}[\![E]\!](A, \eta')$ the following hold:

$$q \neq \bot \tag{7.1}$$

$$(A, \eta[q/\mathrm{v}]) \models Q. \tag{7.2}$$

For example, the Hoare-triple "$P\ \{\mathrm{v} \leftarrow E\}$ true" means that evaluation of E in environments that model P terminates, since every algebra-environment pair models the postcondition "true." If the precondition P is not logically equivalent to "false," then there are no expressions E such that "$P\ \{\mathrm{v} \leftarrow E\}$ false" is valid, because no algebra-environment pair models the assertion "false." However, for all expressions E and all assertions Q, every algebra-environment pair models the Hoare-triple "false $\{\mathrm{v} \leftarrow E\}\ Q$," since the precondition "false" cannot be satisfied.

We say that the Hoare-triple $P\ \{\mathrm{v} \leftarrow E\}\ Q$ is *valid* and write $SPEC \models P\ \{\mathrm{v} \leftarrow E\}\ Q$ when for all *SPEC*-algebras A and for all environments $\eta \in ENV(X, A)$ such that X contains the free identifiers of P and E and Q, η is proper, and η obeys the presumed subtype relation of *SPEC*, $(A, \eta) \models P\ \{\mathrm{v} \leftarrow E\}\ Q$.

Figures 7.1 and 7.2 contain the proof rules for NOAL expressions. In these figures, P, Q, and R are assertions, M is a term, and E, E_1, and so on are NOAL expressions. The notation $\vdash H$, where H is a Hoare-triple, means that one can prove H using the

Figure 7.1: Axiom Schemes for verification of NOAL Expressions.

[ident]	$\vdash$ true $\{\mathtt{v} \leftarrow \mathtt{x}\}$ $M[\mathtt{v}/\mathtt{z}] = M[\mathtt{x}/\mathtt{z}]$	$\mathtt{x}, \mathtt{v}, \mathtt{z} : \mathtt{T}$
[bot]	$\vdash$ false $\{\mathtt{v} \leftarrow \mathtt{bottom[T]}\}$ true	
[ngop]	$\vdash$ true $\{\mathtt{v} \leftarrow \mathtt{T()}\}$ $\mathtt{v} = \mathrm{T}$	
[ginvoc-a]	$\vdash \mathrm{Pre}(\mathtt{g}_{\vec{\mathtt{S}}\to\mathtt{T}})$ $\{\mathtt{y} \leftarrow \mathtt{g}(\vec{\mathtt{x}})\}$ $\mathrm{Post}(\mathtt{g}_{\vec{\mathtt{S}}\to\mathtt{T}})$	$\mathrm{Pre}(\mathtt{g}_{\vec{\mathtt{S}}\to\mathtt{T}})$ observable, $\vec{\mathtt{x}} : \vec{\mathtt{S}}$ and $\mathtt{y} : \mathtt{T}$ formals of spec of $\mathtt{g}_{\vec{\mathtt{S}}\to\mathtt{T}}$
[fcall-a]	$\vdash \mathrm{Pre}(\mathtt{f})$ $\{\mathtt{y} \leftarrow \mathtt{f}(\vec{\mathtt{x}})\}$ $\mathrm{Post}(\mathtt{f})$	$\mathrm{Pre}(\mathtt{f})$ observable, $\vec{\mathtt{x}} : \vec{\mathtt{S}}$ and $\mathtt{y} : \mathtt{T}$ formals of spec of $\mathtt{f}$

proof rules. A proof rule of the form:

$$\frac{h_1, h_2}{c}$$

means that to prove the conclusion c one must first show that both hypotheses h_1 and h_2 hold. Rules written without hypotheses and the horizontal line are axiom schemes.

Each rule is named, for convenience in proofs. The name of a rule appears to the left of that rule. To the right of some of the rules are conditions on types and identifiers. Some of the conditions require an identifier to be *fresh*, which means it is not in the set of free identifiers of either the desired precondition (written P in the figures) or the desired postcondition (Q).

In the following list we discuss each of the rules in Figures 7.1 and 7.2 and explain the conditions that accompany each rule. In the discussion we fix a specification *SPEC* with presumed subtype relation $\leq$. All assertions mentioned are *SPEC*-assertions.

- The rule [ident] is an axiom scheme for all types T, for all identifiers x and v of nominal type T, and for all terms M. The notation $M[\mathtt{x}/\mathtt{z}]$ means M with all free occurrences of z replaced by x.

  The rule says that the only possible result of an expression x is the value of x. Furthermore, it allows one to draw immediate consequences of the equality of v

Figure 7.2: Inference rules for verification of NOAL expressions.

[ginvoc-b] $$\frac{\vdash P\ \{\mathtt{y} \leftarrow \mathtt{(fun\ (}\vec{\mathtt{x}} : \vec{\mathtt{S}}\mathtt{)\ g(}\vec{\mathtt{x}}\mathtt{))\ (}\vec{E}\mathtt{)}\}\ Q}{\vdash P\ \{\mathtt{y} \leftarrow \mathtt{g(}\vec{E}\mathtt{)}\}\ Q} \qquad \vec{E} : \vec{\sigma},\ \vec{\sigma} \leq \vec{\mathtt{S}},\ \sigma_1 = \mathtt{S}_1$$

[fcall-b] $$\frac{\vdash P\ \{\mathtt{y} \leftarrow \mathtt{(fun\ (}\vec{\mathtt{x}} : \vec{\mathtt{S}}\mathtt{)\ f(}\vec{\mathtt{x}}\mathtt{))\ (}\vec{E}\mathtt{)}\}\ Q}{\vdash P\ \{\mathtt{y} \leftarrow \mathtt{f(}\vec{E}\mathtt{)}\}\ Q} \qquad \vec{E} : \vec{\sigma},\ \vec{\sigma} \leq \vec{\mathtt{S}}$$

[comb] $$\frac{\begin{array}{c} \vdash P\ \{\mathtt{v}_1 \leftarrow E_1\}\ R_1, \ldots, \vdash P\ \{\mathtt{v}_n \leftarrow E_n\}\ R_n, \\ \mathtt{v}_1\ \mathcal{R}_{\mathtt{S}_1}\ \mathtt{x}_1 \vdash R_1[\vec{\mathtt{z}}/\vec{\mathtt{x}}] \Rightarrow R_1', \\ \vdots \\ \mathtt{v}_n\ \mathcal{R}_{\mathtt{S}_n}\ \mathtt{x}_n \vdash R_n[\vec{\mathtt{z}}/\vec{\mathtt{x}}] \Rightarrow R_n', \\ \vdash R_1' \,\&\, \cdots \,\&\, R_n'\ \{\mathtt{y}[\vec{\mathtt{z}}/\vec{\mathtt{x}}] \leftarrow E_0\}\ Q[\vec{\mathtt{z}}/\vec{\mathtt{x}}] \end{array}}{\vdash P\ \{\mathtt{y} \leftarrow \mathtt{(fun\ (}\vec{\mathtt{x}} : \vec{\mathtt{S}}\mathtt{)}\ E_0\mathtt{)\ (}E_1, \ldots, E_n\mathtt{)}\}\ Q} \qquad \vec{\mathtt{z}} : \vec{\mathtt{S}} \text{ fresh}$$

[if] $$\frac{\begin{array}{c} \vdash P\ \{\mathtt{v} \leftarrow E_1\}\ \text{true}, \\ \vdash P \,\&\, R_1\ \{\mathtt{v} \leftarrow E_1\}\ \mathtt{v} = \text{true},\ \vdash P \,\&\, R_1\ \{\mathtt{y} \leftarrow E_2\}\ Q, \\ \vdash P \,\&\, R_2\ \{\mathtt{v} \leftarrow E_1\}\ \mathtt{v} = \text{false},\ \vdash P \,\&\, R_2\ \{\mathtt{y} \leftarrow E_3\}\ Q \\ \vdash P \,\&\, R_3\ \{\mathtt{y} \leftarrow E_2\}\ Q,\ \vdash P \,\&\, R_3\ \{\mathtt{y} \leftarrow E_3\}\ Q \\ \vdash (R_1 | R_2 | R_3) = \text{true} \end{array}}{\vdash P\ \{\mathtt{y} \leftarrow \mathtt{if}\ E_1\ \mathtt{then}\ E_2\ \mathtt{else}\ E_3\ \mathtt{fi}\}\ Q}$$

[erratic] $$\frac{\vdash P\ \{\mathtt{y} \leftarrow E_1\}\ Q,\ \vdash P\ \{\mathtt{y} \leftarrow E_2\}\ Q}{\vdash P\ \{\mathtt{y} \leftarrow E_1 \,[]\, E_2\}\ Q}$$

[angelic] $$\frac{\vdash P\ \{\mathtt{y} \leftarrow E_1\}\ Q,\ \vdash P\ \{\mathtt{y} \leftarrow E_2\}\ Q}{\vdash P\ \{\mathtt{y} \leftarrow E_1 \bigtriangledown E_2\}\ Q}$$

[isDef] $$\frac{\vdash P\ \{\mathtt{y} \leftarrow E\}\ \text{true}}{\vdash P\ \{\mathtt{y} \leftarrow \mathtt{isDef?(}E\mathtt{)}\}\ \mathtt{y} = \text{true}}$$

[conseq] $$\frac{\vdash P \Rightarrow P_1,\ \vdash P_1\ \{\mathtt{y} \leftarrow E\}\ Q_1,\ \vdash Q_1 \Rightarrow Q}{\vdash P\ \{\mathtt{y} \leftarrow E\}\ Q} \qquad P, P_1, Q, Q_1 \text{ observable, } \mathtt{y} \text{ not free in } P$$

[carry] $$\frac{\vdash P\ \{\mathtt{y} \leftarrow E\}\ Q}{\vdash P\ \{\mathtt{y} \leftarrow E\}\ P \,\&\, Q} \qquad P \text{ and } Q$$

observable

[rename] $$\frac{\vdash P[\vec{\mathtt{z}}/\vec{\mathtt{x}}]\ \{\mathtt{y}[\vec{\mathtt{z}}/\vec{\mathtt{x}}] \leftarrow E[\vec{\mathtt{z}}/\vec{\mathtt{x}}]\}\ Q[\vec{\mathtt{z}}/\vec{\mathtt{x}}]}{\vdash P\ \{\mathtt{y} \leftarrow E\}\ Q} \qquad \vec{\mathtt{x}} : \vec{\mathtt{T}},\ \vec{\mathtt{z}} : \vec{\mathtt{T}} \text{ fresh}$$

and **x**. This is useful because sometimes "**v** = **x**" is not observable and the rule of consequence (below) does not allow one to use assertions that are not observable.

- The rule [bot] says that the execution of `bottom[T]` never terminates.

- The rule [ngop] is an axiom scheme for all type symbols `T` in the specification *SPEC*. (Recall that `T()` is the desugared form of the expression `T`.) The rule [ngop] says that this invocation returns the class object with the same name.

- The rule [ginvoc-a] is an axiom scheme for all operation specifications of *SPEC*. We denote by Pre($\mathbf{g}_{\vec{S} \to T}$) the precondition of the operation specification named **g** with nominal signature $\vec{S} \to T$. Similarly, Post($\mathbf{g}_{\vec{S} \to T}$) is that operation's postcondition. Notice that the axiom only describes the effect of a generic invocation where the actual argument expressions and the result identifier are exactly the same as the formal arguments and the formal result used in the specification of **g** (with nominal signature $\vec{S} \to T$).

  For soundness, we require that the precondition of this axiom scheme be observable. This restriction ensures that the precondition is meaningful even when the actual arguments do not have the nominal types of the formals.

- The rule [fcall-a] is an axiom scheme for all recursively-defined NOAL functions. The precondition and postcondition of a function come from its specification. As with [ginvoc-a], we require that the precondition of this axiom scheme be observable.

- The inference rule [ginvoc-b] handles the general form of a generic invocation. To prove a triple involving a generic invocation one is obliged to first rewrite the generic invocation from its general form into one where the actual argument expressions are first bound to identifiers. The types of these identifiers must be chosen carefully so that an instance of the rule [ginvoc-a] will apply. However, since we wish to reason based on nominal type information and since the nominal type of the first argument determines the nominal type of the result in NOAL, the nominal type of the first argument cannot be changed. Furthermore, the types of the other arguments are constrained so that the rewritten expression type-checks.

- The inference rule [fcall-b] is like [ginvoc-b] in that it requires one to rewrite a general function call so that the argument expressions are first bound to identifiers. These identifiers should be chosen to match the formal arguments from the specification of the function.

- The inference rule [comb] handles combinations that may include coercions between types. The rule as a whole says that to prove that the desired triple holds, one must first characterize each actual argument expression E_i with some assertion R_i according to the nominal type of E_i. Then one must translate each R_i into an assertion R'_i that describes the actual argument according to the nominal type of the corresponding formal, $\mathbf{x}_i$. This translation uses the axioms that characterize a simulation relation. We will have more to say about such translations below. Finally one must prove that the conjunction of the R'_i is strong enough to show that the possible results of the body of the combination satisfy the desired postcondition.

  The fresh identifiers $\vec{\mathbf{z}}$ are used to hide bindings of $\vec{\mathbf{x}}$ in the assertions so that the proper scope applies. The notation $\mathbf{y}[\vec{\mathbf{z}}/\vec{\mathbf{x}}]$ means $\mathbf{z}_i$ if for some i, $\mathbf{x}_i$ is $\mathbf{y}$. The identifiers $\mathbf{z}_i$ must be fresh to avoid capture problems.

- The inference rule [if] allows one to reason about `if` expressions whose boolean expression (E_1) may be nondeterministic. Before explaining the rule in general we consider two special cases.

  If the boolean expression E_1 is deterministic, let the assertion R_3 be "false" and let R_2 be $\neg R_1$. The assertion R_1 must then characterize the value of E_1 in the following sense. If the desired precondition and R_1 both hold, then the only possible result of E_1 is *true*, otherwise if the desired precondition and $\neg R_1$ both hold, then the only possible result of E_1 is *false*. Then one has to prove that the desired precondition and R_1 together are strong enough to show that the desired postcondition holds on the results of E_2 (the "true" arm) and that the desired precondition and $\neg R_1$ are strong enough to make the postcondition hold on the result of E_3 (the "false" arm). Since R_3 is "false" and $R_2 = \neg R_1$, the other hypotheses follow trivially.

  If the boolean expression E_1 has as possible results bot *true* and *false* when the desired precondition holds (e.g., if E_1 is **`true`** $[]$ **`false`**), then let R_1 and R_2 be "false" and let R_3 be "true." One must then show that E_1 terminates and that the possible results of both E_2 and E_3 satisfy the desired postcondition when the desired precondition holds. The other hypotheses follow trivially.

In general, the assertion R_1 should characterize when E_1 has *true* as its only possible result, R_2 should characterize when E_1 has *false* as its only possible result, and R_3 should characterize when E_1 is nondeterministic. Then one has to show that E_1 terminates, that in each case the postcondition follows, and that all cases are covered. One shows that all cases are covered by showing that $(R_1|R_2|R_3)$ = true.

- The inference rule [erratic] says that the desired postcondition must follow from the desired precondition for each expression.

- The inference rule [angelic] is analogous to the rule [erratic]. Although this rule is sound, it fails to capture all the semantics of angelic choice in NOAL. That is, an angelic choice expression where only one subexpression might fail to terminate would still terminate, but our rule requires that both subexpressions terminate. This incompleteness is caused by the inability of our logic to describe the possible results of an expression separately from its termination.

- The inference rule [isDef] says that to prove that an `isDef?` expression halts (with value *true*), one must prove that all the possible results of the argument expression are proper.

- The general inference rule [conseq] is standard for Hoare logics, where it is often called the "rule of consequence" [Hoa69]. It allows us to use a stronger precondition and a weaker postcondition. The implications that appear in the hypothesis must be provable from the traits of the referenced specification, using the proof rules and axioms of those traits.

  There are two conditions that apply to this rule. First, the preconditions and postconditions m[illegible] observable. This restriction ensures that implication has the expected meaning. Second, the result identifier must not appear free in P; this ensures that the conclusion is well-formed.

- The inference rule [carry] allows us to carry observable assertions from the precondition into the postcondition. We only allow observable assertions to be carried into the postcondition, because only observable properties that hold in an environment are guaranteed to hold when that environment is extended with a possible result and because observability ensures that conjunction has the expected meaning. However, observable preconditions are preserved by expressions, because NOAL is an applicative language.

- The inference rule [rename] allows us to consistently rename identifiers.

A proof in our logic may also use formulas that are provable from the traits of the referenced specification. (Such formulas are used as hypotheses in the rules [conseq] and [if].) These traits always include the trait Bool and the equality trait. The equality trait allows us to interpret the relation "=" that appears in terms as a congruence relation.

Formally, a *proof* of a Hoare-triple for *SPEC* is a list, where the last line in the list is the desired Hoare-triple and each line in the list is either:

- a formula of the form $\vdash Q$, where Q is a *SPEC*-assertion that is provable from the traits of *SPEC*,

- a translation axiom of the form $\mathbf{v}\ \mathcal{R}_{\mathtt{S}}\ \mathbf{x} \vdash P' \Rightarrow Q'$ where P' and Q' are *SPEC*-assertions (as discussed below), or

- a formula of the form $\vdash P'\ \{\mathbf{y} \leftarrow E'\}\ Q'$, where $P'\ \{\mathbf{y} \leftarrow E'\}\ Q'$ is a Hoare-triple for *SPEC*, and the formula either is an axiom or follows from some previous lines by the inference rules of our logic.

We translate assertions about subtypes into assertions about their supertypes using simulation relations. The translations are used in the inference rule [comb], where they appear as formulas of the form:

$$\mathbf{x}\ \mathcal{R}_{\mathtt{S}}\ \mathbf{y} \vdash P(\mathbf{x}) \Rightarrow Q(\mathbf{y}).$$

The above formula means that if the value of $\mathbf{x} : \sigma$ simulates the value of $\mathbf{y} : \mathtt{S}$ at type $\mathtt{S}$ and $P(\mathbf{x})$ holds, then $Q(\mathbf{y})$ holds.

For example, in the proof of a program that referenced the specification IPT we would use the following formulas for translating assertions about instances of `IntTriple` into assertions about instances of `IntPair`:

$$\mathtt{t}\ \mathcal{R}_{\mathbf{IntPair}}\ \mathtt{p} \vdash \mathtt{t}.\text{first} = \mathtt{i} \Rightarrow \mathtt{p}.\text{first} = \mathtt{i} \tag{7.3}$$

$$\mathtt{t}\ \mathcal{R}_{\mathbf{IntPair}}\ \mathtt{p} \vdash \mathtt{t}.\text{second} = \mathtt{i} \Rightarrow \mathtt{p}.\text{second} = \mathtt{i}. \tag{7.4}$$

where $\mathtt{t} : \mathtt{IntTriple}$, $\mathtt{p} : \mathtt{IntPair}$, and $\mathtt{i} : \mathtt{Int}$. These formulas should be thought of as universally quantified over the identifiers involved; that is, the identifier names do not matter.

A proof in our logic may use such a formula as an axiom when the following conditions hold. Let *SPEC* be a fixed specification. Let $\leq$ be the presumed subtype relation of *SPEC* and let *NomSig* be the nominal signature map of *SPEC*. Consider the formula:

$$\mathbf{x}\,\mathcal{R}_{\mathtt{S}}\,\mathbf{y} \vdash P \Rightarrow Q \tag{7.5}$$

where P and Q are *SPEC*-assertions, $\mathbf{x} : \sigma$ does not appear free in Q, and $\mathbf{y} : \mathtt{S}$ does not appear free in P. We take the above formula as an axiom if and only if for all *SPEC*-algebras C, there is some *SPEC*-algebra A, such that there is a simulation relation $\mathcal{R}$ between C and A for *NomSig* and $\leq$ such that $\mathcal{R}$ witnesses $\leq$, and the following property holds for all such A and all such $\mathcal{R}$: for all sets of typed identifiers Z that includes the free identifiers of P and Q except $\mathbf{x} : \sigma$ and $\mathbf{y} : \mathtt{S}$, for all environments $\eta_C \in ENV(Z \cup \{\mathbf{x} : \sigma\}, C)$ and $\eta_A \in ENV(Z \cup \{\mathbf{y} : \mathtt{S}\}, A)$ such that η_C is proper, η_C obeys $\leq$,

$$(C, \eta_C) \quad \models \quad P, \tag{7.6}$$

$$\eta_C(\mathbf{x}) \quad \mathcal{R}_{\mathtt{S}} \quad \eta_A(\mathbf{y}), \tag{7.7}$$

and for each type $\mathtt{T}$ and for each identifier $\mathbf{z} : \mathtt{T} \in Z$,

$$\eta_C(\mathbf{z})\,\mathcal{R}_{\mathtt{T}}\,\eta_A(\mathbf{z}), \tag{7.8}$$

it follows that

$$(A, \eta_A) \models Q. \tag{7.9}$$

As a practical matter, we expect that a small set of axiom schemes should characterize all such translation axioms that one would need to use in a proof. For example, we need two axiom schemes like the ones given above for translating assertions about instances of `IntTriple` into assertions about instances of `IntPair`.

7.2 NOAL Program Verification

In this section we describe how to use the Hoare logic of the previous section to verify NOAL programs. We also give several examples of program verification.

To verify a NOAL program, one first gives specifications for each recursive function definition (see Chapter 4) and shows that the recursive function definitions satisfy their specifications. Then one can use our Hoare logic to show that, if the formals satisfy the desired precondition, the program terminates and the possible results satisfy the desired postcondition. That is, given a program whose body is the expression E, one must prove $\vdash P\ \{\mathtt{v} \leftarrow E\}\ Q$, where `v` is the formal result identifier from the program specification, P is the program's precondition, and Q is its postcondition.

Two steps are required to verify a system of recursive function definitions. First, one shows that for each function `f`, $\vdash$ Pre(`f`) $\{\mathtt{y} \leftarrow E\}$ Post(`f`) follows from the proof rules, where E is the body of `f`, `y` is the formal result identifier from the specification of `f`, Pre(`f`) is the precondition from the specification of `f`, and Post(`f`) is its postcondition. During this proof one can use the axiom schemes [fcall-a] and [fcall-b], which implicitly assume that each recursively defined function meets its specification. This allows one to prove the partial correctness of function bodies containing recursive calls. The second step is to prove that each function terminates whenever it is called with arguments that model its precondition. This step is necessary, since otherwise one could implement a recursive function specification with a body that simply called itself recursively. (We do not provide an explicit method for reasoning about termination.)

As a simple example of function verification, we show that the following function

```
fun sumFirst(p1,p2:IntPair): Int = add(first(p1),first(p2))
```

implements the specification `sumFirst` of Figure 1.4 on page 15 (which references the specification IPT). Since there are no recursive calls, this function always terminates. So *it suffices to show using our* Hoare logic *that*

$$\vdash \text{true}\ \{\texttt{i} \leftarrow \texttt{add(first(p1),first(p2))}\}\ \texttt{i} = \text{p1.first} + \text{p2.second}$$

We find it easiest to start with the formula to be proved as a goal and to use the rules of inference to generate subgoals. Since the body of **sumFirst** consists of a nested generic invocation, the conclusion must follow from the rule [ginvoc-b]. So we rewrite the body of **sumFirst** as follows:

```
(fun (i:Int, j:Int) add(i,j)) (first(p1), first(p2)).
```

(We choose the identifiers `i` and `j` to match the axiom for **add**.)

The rewritten body is a combination, so we must use the rule [comb]. We can see from the rule [comb] that we have as subgoals proving:

$$\vdash \text{true}\ \{\texttt{v1} \leftarrow \texttt{first(p1)}\}\ \texttt{v1} = \text{p1.first} \tag{7.10}$$

$$\vdash \text{true}\ \{\texttt{v2} \leftarrow \texttt{first(p2)}\}\ \texttt{v2} = \text{p2.first} \tag{7.11}$$

$$\texttt{v1}\ \mathcal{R}_{\texttt{Int}}\ \texttt{i} \vdash (\texttt{v1} = \text{p1.first}) \Rightarrow (\texttt{i} = \text{p1.first}) \tag{7.12}$$

$$\texttt{v2}\ \mathcal{R}_{\texttt{Int}}\ \texttt{j} \vdash (\texttt{v2} = \text{p2.first}) \Rightarrow (\texttt{j} = \text{p2.first}) \tag{7.13}$$

$$\vdash \texttt{i} = \text{p1.first}\ \&\ \texttt{j} = \text{p2.first}\ \{\texttt{k} \leftarrow \texttt{add(i,j)}\}\ \texttt{k} = \text{p1.first} + \text{p2.first}. \tag{7.14}$$

The first two subgoals follow from the instance of the axiom scheme [ginvoc-a] for **first**

$$\vdash \text{true}\ \{\texttt{i} \leftarrow \texttt{first(p)}\}\ \texttt{i} = \text{p.first} \tag{7.15}$$

and the inference rule [rename].

The translation axioms for `Int` are valid because if $\mathcal{R}$ is a simulation relation, then $\mathcal{R}_{\texttt{Int}}$ is the identity on integers.

To prove the final subgoal, we use the axiom scheme [ginvoc-a] for **add**:

$$\vdash \text{true}\ \{\texttt{add(i,j)}\}\ \texttt{k} = \texttt{i} + \texttt{j} \tag{7.16}$$

and the rules [conseq] (used twice) and [carry].

Since all the hypotheses of the [comb] rule hold, the desired result follows.

Notice that our proof of the correctness of **sumFirst** does not mention the type `IntTriple`. This is what we mean by using nominal type information in verification.

As an example of program verification, consider the specification **`is2waiting`** given in Figure 4.1 and the program

```
waiting?(m,2).
```

The following lemma verifies that this program is correct. The proof goes into more detail than the previous example.

Lemma 7.2.1. Suppose that $\vdash$ true $\{\texttt{v2} \leftarrow \texttt{2}\}$ $\texttt{v2} = 2$ where `v2` has nominal type `Int`. Suppose further that for all identifiers $\texttt{m}, \texttt{v1}, \texttt{z}$: `Mob` and $\texttt{v2}, \texttt{i}$: `Int`, the following translation axioms are valid:

$$\texttt{v1}\, \mathcal{R}_{\texttt{Mob}}\, \texttt{m} \vdash (2 \in \texttt{z}) = (2 \in \texttt{v1}) \;\Rightarrow\; (2 \in \texttt{z}) = (2 \in \texttt{m}) \tag{7.17}$$

$$\texttt{v2}\, \mathcal{R}_{\texttt{Int}}\, \texttt{i} \vdash \texttt{v2} = 2 \;\Rightarrow\; \texttt{i} = 2 \tag{7.18}$$

Then $\vdash$ true $\{\texttt{b} \leftarrow \texttt{waiting?(m,2)}\}$ $\texttt{b} = (2 \in \texttt{m})$.

Proof: By the inference rule [ginvoc-b], it suffices to show that

$$\vdash \text{true } \{\texttt{b} \leftarrow \texttt{(fun (m:Mob, i:Int) waiting?(m,i)) (m,2)}\}\ \texttt{b} = (2 \in \texttt{m}). \tag{7.19}$$

To show this, we use the rule [comb].

Let R_1 be the assertion $(2 \in \texttt{m}) = (2 \in \texttt{v1})$, where `b` : `Bool` and `v1` : `Mob`. Let R_2 be the assertion $\texttt{v2} = 2$, where `v2` : `Int`.

By the axiom [ident] we have

$$\vdash \text{true } \{\texttt{v1} \leftarrow \texttt{m}\}\ (2 \in \texttt{v1}) = (2 \in \texttt{m}). \tag{7.20}$$

By the hypothesis, we have $\vdash$ true $\{\texttt{v2} \leftarrow \texttt{2}\}$ $\texttt{v2} = 2$. This is the first set of hypotheses for the [comb] rule.

Now we must translate the assertions $(2 \in \texttt{m}) = (2 \in \texttt{v1})$ and $\texttt{v2} = 2$. Since `m` is a free identifier of the precondition and appears free in the first assertion, we rename it to `z`. The second hypothesis of this lemma has the necessary translation axioms.

By the axiom scheme [ginvoc-a] and the specification of `Mob`, we have

$$\vdash \text{true } \{\texttt{b} \leftarrow \texttt{waiting?(m,i)}\}\ \texttt{b} = \texttt{i} \in \texttt{m}. \tag{7.21}$$

Since $(2 \in \mathtt{z}) = (2 \in \mathtt{m})\ \&\ \mathtt{i} = 2 \Rightarrow$ true, by the inference rule [conseq] we have

$$\vdash (2 \in \mathtt{z}) = (2 \in \mathtt{m})\ \&\ \mathtt{i} = 2\ \{\mathtt{b} \leftarrow \mathtt{waiting?(m,i)}\}\ \mathtt{b} = \mathtt{i} \in \mathtt{m}. \tag{7.22}$$

The assertion $(2 \in \mathtt{z}) = (2 \in \mathtt{m})\ \&\ \mathtt{i} = 2$ is observable, since `waiting?(z,2)` is a characteristic observation for $(2 \in \mathtt{z})$, $(2 \in \mathtt{m})$ is similarly observable, and the rest of the assertion can be observed by simple combinations of the operations of types `Bool` and `Int`. So by the inference rule [carry] we can carry the precondition into the postcondition and obtain as our postcondition:

$$(2 \in \mathtt{z}) = (2 \in \mathtt{m})\ \&\ \mathtt{i} = 2\ \&\ \mathtt{b} = \mathtt{i} \in \mathtt{m}. \tag{7.23}$$

Now we use the axioms about equality to show that

$$(2 \in \mathtt{z}) = (2 \in \mathtt{m})\ \&\ \mathtt{i} = 2\ \&\ \mathtt{b} = \mathtt{i} \in \mathtt{m} \Rightarrow \mathtt{b} = (2 \in \mathtt{z}). \tag{7.24}$$

So by a final application of [conseq] we have the desired third hypothesis of the [comb] rule:

$$\vdash (2 \in \mathtt{z}) = (2 \in \mathtt{m})\ \&\ \mathtt{i} = 2\ \{\mathtt{b} \leftarrow \mathtt{waiting?(m,i)}\}\ \mathtt{b} = (2 \in \mathtt{z}), \tag{7.25}$$

Since we have shown all the hypotheses of the [comb] rule, we can conclude that

$$\vdash \text{true}\ \{\mathtt{b} \leftarrow \mathtt{waiting?(m,2)}\}\ \mathtt{b} = (2 \in \mathtt{m}). \tag{7.26}$$

∎

Again, the above proof makes no mention of the subtypes.

To show how our logic handles explicit use of inclusion polymorphism, we will show that

$$\vdash \text{true}\ \{\mathtt{j} \leftarrow \mathtt{sumFirst(make(IntPair,1,2),\ make(IntTriple,4,5,6))}\}\ \mathtt{j} = 5.$$

Since this expression consists of a nested generic invocation, we must use the [ginvoc-b] rule to rewrite our program as follows:

```
(fun (p1,p2:IntPair) sumFirst(p1,p2))
    (make(IntPair,1,2), make(IntTriple,4,5,6)).
```

We now must use the [comb] rule.

It is easy (although tedious) to show that the first set of hypotheses of the [comb] rule hold:

$$\vdash \text{true } \{\texttt{v1} \leftarrow \texttt{make(IntPair,1,2)}\} \text{ v1.first} = 1 \tag{7.27}$$

$$\vdash \text{true } \{\texttt{v2} \leftarrow \texttt{make(IntTriple,4,5,6)}\} \text{ v2.first} = 4, \tag{7.28}$$

where $\texttt{v1} : \texttt{IntPair}$ and $\texttt{v2} : \texttt{IntTriple}$.

We take as translation axioms:

$$\text{v1 } \mathcal{R}_{\texttt{IntPair}} \text{ p1} \vdash (\text{v1.first} = 1) \Rightarrow (\text{p1.first} = 1) \tag{7.29}$$

$$\text{v2 } \mathcal{R}_{\texttt{IntPair}} \text{ p2} \vdash (\text{v2.first} = 4) \Rightarrow (\text{p2.first} = 4). \tag{7.30}$$

We can show the final hypothesis of the [comb] rule using the axiom scheme [fcall-a] and the inference rules [conseq] and [carry]. The axiom for **sumFirst** is:

$$\vdash \text{true } \{\texttt{i} \leftarrow \texttt{sumFirst(p1,p2)}\} \texttt{ i} = \text{p1.first} + \text{p2.first}. \tag{7.31}$$

Using the rules [conseq], [carry], and the trait Int we can conclude:

$$\vdash \text{p1.first} = 1 \,\&\, \text{p2.first} = 4 \; \{\texttt{i} \leftarrow \texttt{sumFirst(p1,p2)}\} \texttt{ i} = 5. \tag{7.32}$$

So by the rule [comb], the desired result follows.

7.3 Soundness of Hoare-style Verification for NOAL

In this section we show the soundness of the Hoare-style symbolic verification technique for NOAL programs described in the previous section.

For soundness to hold we must have simulation relations that witness the referenced specification's presumed subtype relation $\leq$. It is not enough to know $\leq$ is a subtype relation, because we need to ensure that the results of each expression can be simulated at the expression's nominal type. Furthermore, our translation axioms depend on simulation relations.

We also need to assume that certain assertions are observable as required by our Hoare logic. These assumptions ensure that if a precondition holds in an environment that obeys a subtype relation, it also holds in nominal environments that the first environment simulates. We can then find the operation or function's effect using the nominal environment.

The following lemma is the essential step in proving the soundness theorem for our Hoare logic. It says that if some Hoare-triple is provable, and one has an algebra-environment pair that models the precondition, then one can build a nominal algebra-environment pair that models the triple. Soundness follows directly.

The lemma's proof is by induction on the length of proof in our Hoare logic. The interesting cases are the axiom schemes [ginvoca-a], and the rules [comb], [conseq] and [carry] since these are the rules where there is a substantial difference from standard Hoare logics. In the [ginvoc-a] rule one builds a nominal environment that the given environment simulates. Because the precondition is observable, the nominal environment also models the precondition. Since we know what operation is executed by the generic invocation in the nominal environment, we can use the type specification to show that the expression terminates and the postcondition holds for the nominal environment. In the rule [comb] we use the semantics of the translation axioms to build the required nominal environments. In the proofs of the rules [conseq] and [carry] we use the observability of the assertions involved to show that implication and conjunction mean what we expect.

Lemma 7.3.1. Let *SPEC* be a specification. Let $\leq$ be the presumed subtype relation of *SPEC*. Let $\vec{F}$ be a system of mutually recursive NOAL functions. Let C and A be *SPEC*-algebras.

Suppose that $\leq$ is a reflexive and transitive subtype relation on the types of *SPEC* with respect to type-safe NOAL programs, $\leq$ is safe with respect to *NomSig*, each function $\mathtt{f}_i \in \vec{F}$ satisfies its specification, the only type related to each visible type T by $\leq$ is T itself, and $\mathcal{R}$ is a simulation relation between C and A for *NomSig* and $\leq$ such that $\mathcal{R}$ witnesses $\leq$.

Then for all Hoare-triples for *SPEC*, P $\{\mathtt{y} \leftarrow E\}$ Q, *such that the free function* identifiers of E are the $\mathtt{f}_i$, for all sets X of typed identifiers such that X contains all the free identifiers of P and E and Q except y, the following condition holds whenever $\vdash P\ \{\mathtt{y} \leftarrow E\}\ Q$: for all environments $\eta_C \in ENV(X, C)$ that are proper and obey $\leq$, for all nominal environments $\eta_A \in ENV(X, A)$, if $\eta_C\ \mathcal{R}\ \eta_A$ and $(C, \eta_C) \models P$, then $\overline{\eta_A}[P] = true$, and for all possible results $r \in \mathcal{M}[\![E]\!](A, \eta_A)$, $r \neq \bot$, and $(A, \eta_A[r/\mathtt{y}]) \models Q$.

Proof: (by induction on the length of proof in the Hoare logic.)

Let Z be the set of typed function identifiers $\mathtt{f}_i$. To find the set of possible results of a NOAL expression with free function identifiers from Z one needs an environment that is defined on the function identifiers in Z. However, since there is only one denotation for a recursively defined NOAL function in a given algebra, there is only one way to expand a given environment to one that maps the function identifiers in Z to their denotations in that environment's range. Furthermore, if $\eta_C \mathcal{R} \eta_A$, then the expansions of η_C and η_A are also related by $\mathcal{R}$, by Lemma 6.2.2. So to avoid notational complications, we do not mention this expansion below.

We shall often use the following property. If $\eta_C \mathcal{R} \eta_A$ and η_C is proper, then η_A is also proper, since $\mathcal{R}$ is bistrict.

For the basis, we show that the result holds for each of the axiom schemes.

- Suppose the proof consists of an instance of the axiom scheme [ident]

$$\vdash \text{true } \{\mathtt{v} \leftarrow \mathtt{x}\}\ M[\mathtt{v}/\mathtt{z}] = M[\mathtt{x}/\mathtt{z}]$$

  for some identifiers $\mathtt{x}$, $\mathtt{v}$, and $\mathtt{z}$ of some type $\mathtt{T}$. Let X be a set of typed identifiers that includes $\mathtt{x} : \mathtt{T}$ but not $\mathtt{v} : \mathtt{T}$. Let $\eta_C \in ENV(X, C)$ be proper and obey $\leq$. Let $\eta_A \in ENV(X, A)$ be a nominal environment such that $\eta_C \mathcal{R} \eta_A$. It is trivial that $(C, \eta_C) \models$ true and that $\overline{\eta_A}[\![\text{true}]\!] = \mathit{true}$. By definition, $\mathcal{M}[\![\mathtt{x}]\!](A, \eta_A) = \{\eta_A(\mathtt{x})\}$. Since η_C is proper $\eta_A(\mathtt{x})$ is also proper. Finally, it is trivial that $(A, \eta_A[\eta_A(\mathtt{x})/\mathtt{v}]) \models M[\mathtt{v}/\mathtt{z}] = M[\mathtt{x}/\mathtt{z}]$.

- Suppose the proof consists of an instance of the axiom scheme [bot]

$$\vdash \text{false } \{\mathtt{v} \leftarrow \mathtt{bottom[T]}\} \text{ true.}$$

  Then the result follows trivially, since no environment can model the precondition "false."

- Suppose the proof consists of an instance of the axiom scheme [ngop]

$$\vdash \text{true } \{\mathtt{v} \leftarrow \mathtt{T()}\}\ \mathtt{v} = \mathtt{T}$$

  where $\mathtt{v}$ has nominal type $\mathtt{TClass}$. (This return type is uniquely determined by $\mathtt{T}$, because of the restrictions of our specification language.) Let X be a set of typed identifiers that does not contain v : TClass. Let $\eta_C \in ENV(X,C)$ be proper and obey $\leq$. Let $\eta_A \in ENV(X,A)$ be a nominal environment such that $\eta_C \mathcal{R} \eta_A$. It is trivial that $(C,\eta_C) \models$ true and that $\overline{\eta_A}[\![\text{true}]\!] = \textit{true}$. By definition of when an algebra satisfies a specification (see Chapter 2), the only possible result of $\mathcal{M}[\![\texttt{T()}]\!](A,\eta_A)$ is the proper object of type TClass that is the result of the trait function "T," call it T_A. So $(A,\eta_A[T_A/\mathtt{v}]) \models \mathtt{v} = \mathrm{T}$.

- Suppose the proof consists of an instance of the axiom scheme [ginvoc-a]

$$\vdash \mathrm{Pre}(\mathtt{g}_{\vec{\mathtt{S}}\to\mathtt{T}})\ \{\mathtt{y} \leftarrow \mathtt{g}(\vec{\mathtt{x}})\}\ \mathrm{Post}(\mathtt{g}_{\vec{\mathtt{S}}\to\mathtt{T}})$$

where the $\vec{\mathtt{x}} : \vec{\mathtt{S}}$ are the formal arguments from the specification of the operation g with nominal signature $\vec{\mathtt{S}} \to \mathtt{T}$ and y : T is the formal result identifier from that specification of g.

By the restrictions of our specification language, the types of the formal arguments uniquely determine the type of the formal result and hence the precondition and postcondition. That is, the specification *SPEC* only contains one operation specification for g with this nominal signature. This means that *SPEC*-algebras have an operation named $\mathtt{g}_{\vec{\mathtt{S}}\to\mathtt{T}}$, but they cannot have operations named $\mathtt{g}_{\vec{\mathtt{S}}\to\tau}$ for $\tau \neq \mathtt{T}$. Let X contain the formals $\mathtt{x}_i$ but not y : T. Let $\eta_C \in ENV(X,C)$ be proper and obey $\leq$. Let $\eta_A \in ENV(X,A)$ be a nominal environment such that $\eta_C \mathcal{R} \eta_A$. Suppose that $(C,\eta_C) \models \mathrm{Pre}(\mathtt{g}_{\vec{\mathtt{S}}\to\mathtt{T}})$. Since by hypothesis this precondition is observable, $(A,\eta_A) \models \mathrm{Pre}(\mathtt{g}_{\vec{\mathtt{S}}\to\mathtt{T}})$ by Lemma 5.2.4. Since η_A is a nominal environment, by Lemma 4.4.3

$$\overline{\eta_A}[\![\mathrm{Pre}(\mathtt{g}_{\vec{\mathtt{S}}\to\mathtt{T}})]\!] = \textit{true}. \tag{7.33}$$

Since η_A is a nominal environment,

$$\mathcal{M}[\![\mathtt{g}(\vec{\mathtt{x}})]\!](A,\eta_A) = A_{\mathtt{g}_{\vec{\mathtt{S}}\to\mathtt{T}}}(\eta(\vec{\mathtt{x}})). \tag{7.34}$$

By definition of when an operation satisfies its specification, for all possible results $r \in A_{\mathtt{g}_{\vec{\mathtt{S}}\to\mathtt{T}}}(\eta(\vec{\mathtt{x}}))$, $r \neq \bot$ and $\overline{\eta_A[r/\mathtt{y}]}[\![\mathrm{Post}(\mathtt{g}_{\vec{\mathtt{S}}\to\mathtt{T}})]\!] = \textit{true}$.

- Suppose the proof consists of an instance of the axiom scheme [fcall-a]

$$\vdash \mathrm{Pre}(\mathtt{f})\ \{\mathtt{y} \leftarrow \mathtt{f}(\vec{\mathtt{x}})\}\ \mathrm{Post}(\mathtt{f})$$

where the $\vec{\mathtt{x}} : \vec{\mathtt{S}}$ are the formal arguments from the specification of the NOAL function `f` with nominal signature $\vec{\mathtt{S}} \rightarrow \mathtt{T}$ and where $\mathtt{y} : \mathtt{T}$ is the formal result identifier from this specification. Let X contain the formals $\mathtt{x}_i$ but not $\mathtt{y} : \mathtt{T}$. Let $\eta_C \in ENV(X, C)$ be proper and obey $\leq$. Let $\eta_A \in ENV(X, A)$ be a nominal environment such that $\eta_C \mathcal{R} \eta$.

Suppose that $(C, \eta_C) \models \mathrm{Pre}(\mathtt{f})$. Since by hypothesis the precondition is observable, $(A, \eta_A) \models \mathrm{Pre}(\mathtt{f})$ by Lemma 5.2.4. So by Lemma 4.4.3, $\overline{\eta}[\![\mathrm{Pre}(\mathtt{f})]\!] = \mathit{true}$.

By hypothesis, `f` satisfies its specification. So by definition of when a NOAL function satisfies its specification, for all possible results $r \in \mathcal{M}[\![\mathtt{f}(\vec{\mathtt{x}})]\!](A, \eta_A)$, $r \neq \bot$ and $(A, \eta_A[r/\mathtt{y}]) \models \mathrm{Post}(\mathtt{f})$.

For the inductive step, we suppose that the result holds for all proofs of length less than n. Suppose that we are given a proof of length $n > 1$. The last step of the proof must be either an axiom or the conclusion of an inference rule in our proof system. The axioms were covered above, so it remains to deal with each of the rules of inference. Since all the conclusions of the rules of inference have a similar form ($\vdash P \; \{\mathtt{y} \leftarrow E\} \; Q$) we establish some conventions here instead of repeating them in what follows. Let the nominal type of the expression (E) be T. Let X be a set of typed identifiers that contains all the free identifiers of P, E and Q except the result identifier $\mathtt{y} : \mathtt{T}$. Let $\eta_C \in ENV(X, C)$ be proper and obey $\leq$. Let $\eta_A \in ENV(X, A)$ be a nominal environment such that $\eta_C \mathcal{R} \eta_A$.

- Suppose the last step is the conclusion of the rule [ginvoc-b]:

$$\vdash P \; \{\mathtt{y} \leftarrow \mathtt{g}(\vec{E})\} \; Q.$$

The hypothesis of this rule

$$\vdash P \; \{\mathtt{y} \leftarrow \mathtt{(fun} \;\; (\vec{\mathtt{x}} : \vec{\mathtt{S}}) \;\; \mathtt{g}(\vec{\mathtt{x}})) \;\; (\vec{E})\} \; Q$$

must therefore appear in an earlier step. Since by definition of NOAL,

$$\mathcal{M}[\![\mathtt{g}(\vec{E})]\!](C, \eta_C) = \mathcal{M}[\![\mathtt{(fun} \;\; (\vec{\mathtt{x}} : \vec{\mathtt{S}}) \;\; \mathtt{g}(\vec{\mathtt{x}})) \;\; (\vec{E})] \;\;](C, \eta_C \qquad (7.35)$$

the result follows.

- Suppose the last step is the conclusion of the rule [fcall-b]. Then the claim follows as for the rule [ginvoc-b].

- Suppose the last step is the conclusion of the rule [comb]:

$$\vdash P \ \{\mathbf{y} \leftarrow (\mathbf{fun}\ (\vec{\mathbf{x}} : \vec{\mathbf{S}})\ \mathbf{g}(\vec{\mathbf{x}}))\ (E_1, \ldots, E_n)\}\ Q.$$

Suppose $(C, \eta_C) \models P$.

For each i from 1 to n, there is some earlier step in the proof of the form $\vdash P\ \{\mathbf{v}_i \leftarrow E_i\}\ R_i$. Let $r_i \in \mathcal{M}[\![E_i]\!](A, \eta_A)$ be a possible result of the argument expression E_i, whose nominal type is σ_i. By the inductive hypothesis,

$$\overline{\eta_A}[\![P]\!] = \mathit{true}, \tag{7.36}$$

$r_i \neq \bot$, and $(A, \eta_A[r_i/\mathbf{v}_i]) \models R_i$. By Lemma 6.2.2 and Lemma 6.2.1, each $q_i \in \mathcal{M}[\![E_i]\!](C, \eta_C)$ is such that $q_i\ \mathcal{R}\ r_i$.

There must also be some step in the proof of the form

$$\mathbf{v}_i\ \mathcal{R}_{\mathbf{S}_i}\ \mathbf{x}_i \vdash R_i[\vec{\mathbf{z}}/\vec{\mathbf{x}}] \Rightarrow R_i' \tag{7.37}$$

To use this translation axiom we show how suitable versions of the above environments satisfy the conditions for a translation axiom. We define the following environments:

$$\eta_C' \stackrel{\text{def}}{=} \eta_C[\eta_C(\vec{\mathbf{x}})/\vec{\mathbf{z}}] \tag{7.38}$$

$$\eta_A' \stackrel{\text{def}}{=} \eta_A[\eta_A(\vec{\mathbf{x}})/\vec{\mathbf{z}}]. \tag{7.39}$$

By construction, $\eta_C'\ \mathcal{R}\ \eta_A'$. By the above, for each i, $(C, \eta_C'[q_i/\mathbf{v}_i]) \models R_i[\vec{\mathbf{z}}/\vec{\mathbf{x}}]$ and the environment $\eta_C'[q_i/\mathbf{v}_i]$ is proper. Since $\leq$ is reflexive and transitive, $\leq$ is safe with respect to *NomSig*, and E_i is type-safe, the environment $\eta_C'[q_i/\mathbf{v}_i]$ obeys $\leq$ by Lemma 3.6.5. So by the translation axiom quoted in the proof, $(A, \eta_A'[r_i/\mathbf{x}_i]) \models R_i'$.

By definition of NOAL, r is a possible result of the entire combination expression only if $r \in \mathcal{M}[\![E_0]\!](A, \eta_A'[\vec{r}/\vec{\mathbf{x}}])$. There must be a step in the proof of the form

$$\vdash R_1' \ \&\ \cdots\ \&\ R_n'\ \{\mathbf{y}' \leftarrow E_0\}\ Q[\vec{\mathbf{z}}/\vec{\mathbf{x}}], \tag{7.40}$$

where $\mathbf{y}'$ is $\mathbf{y}[\vec{\mathbf{z}}/\vec{\mathbf{x}}]$. By construction,

$$(A, \eta'_A[\vec{r}/\vec{\mathbf{x}}]) \models R'_1 \,\&\, \cdots \,\&\, R'_n. \tag{7.41}$$

Since $\eta'_C[\vec{q}/\vec{\mathbf{x}}] \;\mathcal{R}\; \eta'_A[\vec{r}/\vec{\mathbf{x}}]$, by Lemma 5.2.3,

$$(C, \eta'_C[\vec{q}/\vec{\mathbf{x}}]) \models R'_1 \,\&\, \cdots \,\&\, R'_n. \tag{7.42}$$

By the inductive hypothesis,

$$r \neq \bot \tag{7.43}$$

and $(A, (\eta'_A[\vec{r}/\vec{\mathbf{x}}])[r/\mathbf{y}']) \models Q[\vec{\mathbf{z}}/\vec{\mathbf{x}}]$. So by undoing the renamings, we have

$$(A, (\eta_A[\vec{r}/\vec{\mathbf{z}}])[r/\mathbf{y}]) \models Q. \tag{7.44}$$

Since the $\mathbf{z}$ are fresh,

$$(\eta_A[r/\mathbf{y}])[\vec{r}/\vec{\mathbf{z}}] = (\eta_A[\vec{r}/\vec{\mathbf{z}}])[r/\mathbf{y}], \tag{7.45}$$

and therefore

$$(A, (\eta_A[r/\mathbf{y}])[\vec{r}/\vec{\mathbf{z}}]) \models Q. \tag{7.46}$$

Since the $\vec{\mathbf{z}}$ do not occur free in Q, by Lemma 5.2.6,

$$(A, \eta_A[r/\mathbf{y}]) \models Q. \tag{7.47}$$

- Suppose the last step is the conclusion of the rule [if]:

  $$\vdash P \; \{\mathtt{y} \leftarrow \mathtt{if}\ E_1\ \mathtt{then}\ E_2\ \mathtt{else}\ E_3\ \mathtt{fi}\} \; Q$$

  Suppose $(C, \eta_C) \models P$.

  There must be an earlier step in the proof of the form $\vdash P\ \{\mathtt{v} \leftarrow E_1\}$ true. So by the inductive hypothesis,

  $$\overline{\eta_A}[\![P]\!] = \mathit{true}, \tag{7.48}$$

  and for all $r_1 \in \mathcal{M}[\![E_1]\!](A, \eta_A)$, $r_1 \neq \bot$. Since E_1 has nominal type `Bool` and no other types are related to `Bool` by $\leq$, each r_1 must have type `Bool`.

  There must be an earlier step in the proof of the form $\vdash P \,\&\, R_1\ \{\mathtt{v} \leftarrow E_1\}\ \mathtt{v} = \text{true}$. By the inductive hypothesis, if $(C, \eta_C) \models P \,\&\, R_1$, then $\mathcal{M}[\![E_1]\!](A, \eta_A) = \{\mathit{true}\}$.

Furthermore, there must be an earlier step in the proof of the form $\vdash P \,\&\, R_1 \ \{\mathrm{y} \leftarrow E_2\}\ Q$. So if $(C, \eta_C) \models P \,\&\, R_1$, then for all $r_2 \in \mathcal{M}[\![E_2]\!](A, \eta_A)$, $r_2 \neq \bot$ and $(A, \eta_A[r_2/\mathrm{y}]) \models Q$.

There must also be earlier steps in the proof of the form $\vdash P \,\&\, R_2 \ \{\mathrm{v} \leftarrow E_1\}\ \mathrm{v} =$ false and $\vdash P \,\&\, R_2 \ \{\mathrm{v} \leftarrow E_3\}\ Q$, As above, if $(C, \eta_C) \models P \,\&\, R_2$, then for all $r_3 \in \mathcal{M}[\![E_3]\!](A, \eta_A)$, $r_3 \neq \bot$ and $(A, \eta_A[r_3/\mathrm{y}]) \models Q$.

There must also be earlier steps in the proof of the form $\vdash P \,\&\, R_3 \ \{E_2\}\ Q$ and $\vdash P \,\&\, R_3 \ \{E_3\}\ Q$. So if $(C, \eta_C) \models P \,\&\, R_3$, then for all $r_{23} \in \mathcal{M}[\![E_2]\!](A, \eta_A) \cup \mathcal{M}[\![E_3]\!](A, \eta_A)$, $r_{23} \neq \bot$ and $(A, \eta_A[r_{23}/\mathrm{y}]) \models Q$.

Finally, there must be an earlier step in the proof of the form $(R_1 | R_2 | R_3) =$ true. Since $(C, \eta_C) \models P$, it follows that for some i from 1 to 3, $(C, \eta_C) \models P \,\&\, R_i$. So by the inductive hypothesis, $\overline{\eta_A}[\![P \,\&\, R_i]\!] = \mathit{true}$. Let r be a possible result of the `if` expression in (A, η_A); that is suppose that

$$r \in \bigcup_{r_1 \in \mathcal{M}[\![E_1]\!](A, \eta_A)} \begin{cases} \mathcal{M}[\![E_2]\!](A, \eta_A) & \text{if } r_1 = \mathit{true} \\ \mathcal{M}[\![E_3]\!](A, \eta_A) & \text{if } r_1 = \mathit{false} \\ \{\bot\} & \text{otherwise} \end{cases} \qquad (7.49)$$

Then $r \neq \bot$, because by the above, the only possible results of E_1 are either *true* or *false*. Furthermore, for each i such that $\overline{\eta_A}[\![P \,\&\, R_i]\!] = \mathit{true}$, $r \neq \bot$ and $(A, \eta_A[r/\mathrm{y}]) \models Q$ by the above.

- Suppose the last step is the conclusion of the rule [erratic]:

$$\vdash P \ \{\mathrm{y} \leftarrow E_1 \,[\!]\, E_2\}\ Q$$

Suppose $(C, \eta_C) \models P$. There must be earlier steps in the proof of the form $\vdash P \ \{\mathrm{y} \leftarrow E_1\}\ Q$ and $\vdash P \ \{\mathrm{y} \leftarrow E_2\}\ Q$. Each possible result of $E_1 \,[\!]\, E_2$ is a possible result of either E_1 or E_2. By the inductive hypothesis, $\overline{\eta_A}[\![P]\!] = \mathit{true}$, and for each possible result r of either E_1 or E_2 in (A, η_A), $r \neq \bot$ and $(A, \eta_A[r/\mathrm{y}]) \models Q$.

- Suppose the last step is the conclusion of the rule [angelic]. Then the result follows as for the rule [erratic], since each possible result of $E_1 \bigtriangledown E_2$ is a possible result of either E_1 or E_2.

- Suppose the last step is the conclusion of the rule [isDef]:

$$\vdash P\ \{\mathtt{y} \leftarrow \mathtt{isDef?}(E)\}\ \mathtt{y} = \mathrm{true}$$

Suppose $(C, \eta_C) \models P$. There must be an earlier step in the proof of the form $\vdash P\ \{\mathtt{y} \leftarrow E\}$ true. By the inductive hypothesis, $\overline{\eta_A}[\![P]\!] = \mathit{true}$ and for all $r \in \mathcal{M}[\![E]\!](A, \eta_A)$, $r \neq \bot$. Therefore, by definition of NOAL, $\mathcal{M}[\![\mathtt{isDef?}(E)]\!](A, \eta_A) = \{\mathit{true}\}$. Furthermore, since *true* is the only possible result it only remains to show that $(A, \eta_A[\mathit{true}/\mathtt{y}]) \models \mathtt{y} = \mathrm{true}$, which is trivially true.

- Suppose the last step is the conclusion of the rule [conseq]:

$$\vdash P\ \{\mathtt{y} \leftarrow E\}\ Q$$

Suppose $(C, \eta_C) \models P$. Since P is observable, by Lemma 5.2.4 and Lemma 4.4.3, $\overline{\eta_A}[\![P]\!] = \mathit{true}$.

There must be a step in the proof of the form $\vdash P \Rightarrow P_1$. In general, P_1 and Q_1 may have more free identifiers than P and Q. For example, if $\mathtt{i}$ has nominal type integer, then the formula "true $\Rightarrow \mathtt{i} = \mathtt{i}$" and its converse are both valid. Let $\vec{\mathtt{z}} : \vec{\mathtt{S}}$ be a tuple of all the free identifiers of P_1 and Q_1 except for the result identifier $\mathtt{y} : \mathtt{T}$ that are not in X (i.e., that are not in the domain of η_C). Let $\vec{q} \in C_{\vec{\mathtt{S}}}$ be a tuple of proper elements. Since $\leq$ is reflexive, $\eta_C[\vec{q}/\vec{\mathtt{z}}]$ obeys $\leq$. Since $\mathcal{R}$ witnesses $\leq$, there are $\vec{r} \in A_{\vec{\mathtt{S}}}$ such that

$$\eta_C[\vec{q}/\vec{\mathtt{z}}]\ \mathcal{R}\ \eta_A[\vec{r}/\vec{\mathtt{z}}]. \tag{7.50}$$

Notice that $\eta_A[\vec{r}/\vec{\mathtt{z}}]$ is nominal. Since $P \Rightarrow P_1$ is valid in all nominal environments,

$$\overline{\eta_A[\vec{r}/\vec{\mathtt{z}}]}[\![P \Rightarrow P_1]\!] = \mathit{true}. \tag{7.51}$$

So by definition,

$$(C, \eta_C[\vec{q}/\vec{\mathtt{z}}]) \models P \Rightarrow P_1. \tag{7.52}$$

Therefore, by Theorem 5.2.5,

$$(C, \eta_C[\vec{q}/\vec{\mathtt{z}}]) \models P_1. \tag{7.53}$$

There must also be a step in the proof of the form $\vdash P_1 \; \{\mathrm{y} \leftarrow E\} \; Q_1$. By the inductive hypothesis and the above,

$$\overline{\eta_A[\vec{r}/\vec{z}]}[\![P_1]\!] = \mathit{true} \tag{7.54}$$

and for all $r' \in \mathcal{M}[\![E]\!](A, \eta_A[\vec{r}/\vec{z}])$, $r' \neq \perp$, and $(A, (\eta_A[\vec{r}/\vec{z}])[r'/\mathrm{y}]) \models Q_1$.

Finally, there must be an earlier step in the proof of the form $\vdash Q_1 \Rightarrow Q$. Since E is type-safe, $\leq$ is reflexive and transitive, and $\leq$ is safe with respect to *NomSig*, by Lemma 3.6.5 $(\eta_A[\vec{r}/\vec{z}])[r'/\mathrm{y}]$ obeys $\leq$. Since Q_1 and Q are observable, by Theorem 5.2.5

$$(A, (\eta_A[\vec{r}/\vec{z}])[r'/\mathrm{y}]) \models Q \tag{7.55}$$

Since the $\mathbf{z}_i$ are not free in Q, by Lemma 5.2.6,

$$(A, \eta_A[r'/\mathrm{y}]) \models Q. \tag{7.56}$$

- Suppose the last step is the conclusion of the rule [carry]:

$$\vdash P \; \{\mathrm{y} \leftarrow E\} \; P \,\&\, Q$$

Suppose $(C, \eta_C) \models P$. There must be an earlier step in the proof of the form $\vdash P \; \{\mathrm{y} \leftarrow E\} \; Q$. By the inductive hypothesis, $\overline{\eta_A}[\![P]\!] = \mathit{true}$ and for all $r \in \mathcal{M}[\![E]\!](A, \eta_A)$, $r \neq \perp$ and

$$(A, \eta_A[r/\mathrm{y}]) \models Q. \tag{7.57}$$

Since $\overline{\eta_A}[\![P]\!] = \mathit{true}$ and η_A is nominal, $(A, \eta_A) \models P$. Since $\leq$ is reflexive and transitive and $\leq$ is safe with respect to *NomSig*, by Lemma 3.6.5, $\eta_A[r/\mathrm{y}]$ obeys $\leq$. Since y is not free in P, $\leq$ is a subtype relation, $\eta_A[r/\mathrm{y}]$ obeys $\leq$ and P is observable, by Lemma 5.2.7,

$$(A, \eta_A[r/\mathrm{y}]) \models P. \tag{7.58}$$

Since $P \,\&\, Q$ is logically equivalent to $\neg(P \Rightarrow \neg Q)$, P and Q are observable, and $\eta_A[r/\mathrm{y}]$ is proper and obeys $\leq$, by Theorem 5.2.2 and Theorem 5.2.5,

$$(A, \eta_A[r/\mathrm{y}]) \models P \,\&\, Q. \tag{7.59}$$

- Suppose the last step is the conclusion of the rule [rename]:

$$\vdash P\ \{\mathtt{y} \leftarrow E\}\ Q$$

Since the identifiers $\vec{\mathbf{z}}$ are fresh, the possible results of $E[\vec{\mathbf{z}}/\vec{\mathbf{x}}]$ in $\eta_C[\eta_C(\vec{\mathbf{x}})/\vec{\mathbf{z}}]$ are the same as the possible results of E in (C, η_C).

∎

The following theorem is the soundness theorem for our Hoare logic.

Theorem 7.3.2. Let *SPEC* be a specification. Let $\leq$ be the presumed subtype relation of *SPEC*. Let *NomSig* be the nominal signature map of *SPEC*. Let $\vec{F}$ be a system of mutually recursive NOAL functions.

Suppose that $\leq$ is reflexive and transitive, $\leq$ is safe with respect to *NomSig*, the only type related to each visible type $\mathtt{T}$ by $\leq$ is $\mathtt{T}$ itself, each function $\mathbf{f}_i \in \vec{F}$ satisfies its specification, and that for every *SPEC*-algebra C there is some *SPEC*-algebra A and some simulation relation $\mathcal{R}$ between C and A for *NomSig* and $\leq$ such that $\mathcal{R}$ witnesses $\leq$. Then for all Hoare-triples $P\ \{\mathtt{y} \leftarrow E\}\ Q$ for *SPEC*, if the free function identifiers of E are the $\mathbf{f}_i$ in $\vec{F}$ and $\vdash P\ \{\mathtt{y} \leftarrow E\}\ Q$, then $SPEC \models P\ \{\mathtt{y} \leftarrow E\}\ Q$.

Proof: Let $P\ \{\mathtt{y} \leftarrow E\}\ Q$ be a Hoare-triple, where X is the set of free identifiers of P and the nominal type of E is some type $\mathtt{T}$. Let C be a *SPEC*-algebra. Let $\eta_C \in ENV(X, C)$ be such that η_C is proper and obeys $\leq$. Suppose that $(C, \eta_C) \models P$.

By hypothesis, there is some *SPEC*-algebra A and some simulation relation $\mathcal{R}$ between C and A for *NomSig* and $\leq$ such that $\mathcal{R}$ witnesses $\leq$. Using this $\mathcal{R}$ we can build a nominal environment $\eta_A \in ENV(X, A)$ such that $\eta_C\ \mathcal{R}\ \eta_A$. By Lemma 7.3.1, $\overline{\eta_A}[\![P]\!] = \mathit{true}$, and for all $r \in \mathcal{M}[\![E]\!](A, \eta_A)$, $r \neq \bot$ and $(A, \eta_A[r/\mathtt{y}]) \models Q$. Since η_A is nominal, $(A, \eta_A) \models P$. Since $\mathcal{R}$ is a simulation relation, by Lemma 6.2.2 and Lemma 6.2.1, for each $q \in \mathcal{M}[\![E]\!](C, \eta_C)$, there is some $r' \in \mathcal{M}[\![E]\!](A, \eta_A)$ such that $q\,\mathcal{R}_{\mathtt{T}}\,r'$. So q is proper, since $\mathcal{R}$ is bistrict. Since $\eta_C[q/\mathtt{y}]\ \mathcal{R}\ \eta_A[r/\mathtt{y}]$, by Lemma 5.2.3, $(C, \eta_C[q/\mathtt{y}]) \models Q$. ∎

In the above theorem, we assumed that each recursively defined function appearing in a program meets its specification. Since our technique for proving the partial correctness of such functions is standard, we will not show formally that it is sound.

The following corollary says that our technique for the verification of programs is sound. This is not a trivial consequence of the soundness theorem because the semantics of our specifications require that each possible result be an instance of a subtype of the program specification's nominal return type. So in the following corollary we combine the soundness theorem with Lemma 3.6.5, which says that the possible results of a type-safe NOAL expression must be instances of subtypes of the expression's nominal type. This connection to the type system of NOAL is explored further in the next section.

Corollary 7.3.3. Let p be a program specification, with referenced specification *SPEC*, precondition R and postcondition Q and nominal result type S. Let $\leq$ be the presumed subtype relation of *SPEC*. Let *NomSig* be the nominal signature map of *SPEC*. Let $P = \vec{F}; E$ be a NOAL program with nominal type S, where $\vec{F}$ is a system of mutually recursive NOAL functions, and E is a NOAL expression.

Suppose that $\leq$ is reflexive and transitive, $\leq$ is safe with respect to *NomSig*, the only type related to each visible type T by $\leq$ is T itself, each function $\mathtt{f}_i \in \vec{F}$ satisfies its specification, and that for every *SPEC*-algebra C there is some *SPEC*-algebra A and some simulation relation $\mathcal{R}$ between C and A for *NomSig* and $\leq$ such that $\mathcal{R}$ witnesses $\leq$. If $\vdash R \; \{\mathtt{y} \leftarrow E\} \; Q$, then P satisfies the specification p.

Proof: By the previous theorem, $SPEC \models R \; \{\mathtt{y} \leftarrow E\} \; Q$. Since E has nominal type S, by Lemma 3.6.5, each possible result of E has some type $\sigma \leq \mathtt{S}$. So by definition, P satisfies the specification p. ∎

7.4 How a Type System can Aid Verification

In this section we discuss how type checking can be used to aid program verification.

7.4.1 Obedience in NOAL

For soundness of our Hoare-style verification technique for NOAL, we must ensure that the possible results of each expression are instances of a subtype of the expression's nominal type. We call this property *obedience*. Fundamentally, obedience is necessary because our function specifications require that the arguments and results of a function obey a subtype relation. Obedience also ensures that one's reasoning about assertions using standard techniques such as the proof by contradiction is sound, as we showed in Chapter 5. The soundness proof for our Hoare logic requires obedience to a subtype relation for the same reason.

Instead of checking obedience with our Hoare logic, we separate type checking from the rest of the verification problem. Separating type checking from our Hoare logic allows the logic to be simpler than it would be otherwise. Furthermore, this separation also allows us to use a computer to check mechanically whether our program is type-safe, removing part of the burden of program verification from humans.

The NOAL type system can ensure obedience of type-safe expressions over a specification *SPEC* if the following conditions are met.

- The specification's presumed subtype relation must be reflexive.
- The specification's presumed subtype relation must be transitive.
- The presumed subtype relation must be safe with respect to the specification's nominal signature map.

These reasons for these conditions are explained in Section 3.6.

In addition, the above conditions are plausible and intuitive. Unless $\leq$ is reflexive, nominal environments will not be obedient, making our reasoning based on nominal environments counterintuitive. Transitive relations are also more compactly described than nontransitive ones, because one can just describe a basis for the relation and take the transitive closure. Finally, unless $\leq$ is safe, a "subtype" may not have all the instance operations of its supertypes, and so its instances could not be used everywhere that instances of the supertype could be used.

7.4.2 Verification in Trellis/Owl

It is easiest to use our Hoare-style verification techniques in a statically typed object-oriented programming language, because a properly designed type system can do some of the verification work automatically, as in NOAL. One such language is Trellis/Owl [SCB*86], whose type system was the inspiration for the NOAL type system. Like NOAL, Trellis/Owl type system is based on nominal signatures and a presumed subtype relation.

Trellis/Owl limits presumed subtype relations to be partial orders, that is reflexive, transitive, and antisymmetric relations on types. Although reflexive and transitive relations are necessary for type-checking and verification, antisymmetry is not. Trellis/Owl requires that a presumed subtype relation be antisymmetric, because the implementation of each presumed subtype is also a subclass, and cyclic inheritance relationships are nonsensical. However, symmetric subtype relationships are useful. For example, consider types `HashTable` and `BTree`. We can specify these types so instances of these types obey a common protocol for inserting and finding elements and so that each is a subtype of the other, although they can have different class operations.

The Trellis/Owl type system supports program verification by ensuring obedience to the presumed subtype relation. As in NOAL, it ensures this by requiring that the presumed subtype relation is safe, reflexive and transitive. The presumed subtype relation of a Trellis/Owl program is declared, so that if the presumed subtype relation is a subtype relation (in our sense), then our style of reasoning should be useful for program verification in Trellis/Owl.

7.4.3 Verification in Emerald

Unlike Trellis/Owl the designers of Emerald [BHJL86] have made the mistake of inferring subtype relationships for abstract types. Unfortunately, it is easy to specify types so that the binary relation that Emerald infers is not a subtype relation in our sense. Therefore to ensure that every environment obeys a subtype relation in an Emerald program, one has to duplicate work that the type checker could have done.

7.4.4 Verification in Smalltalk-80

Many popular languages, such as Smalltalk-80 [GR83], are not statically type-checked but are type-checked dynamically. In Smalltalk-80, type information is not checked during assignments, but only on generic invocation. To support data abstraction, Smalltalk-80 ensures that each object is manipulated only by the instance operations defined by its class (including those inherited from superclasses). Therefore, when an instance operation named g is invoked on an object q, the class that implements q must define operation g; if it does not, an error occurs and is reported to the user. Type information about objects is also available to Smalltalk-80 programs during execution.

To use our Hoare-style reasoning techniques on a Smalltalk-80 program, we need a notion of nominal type and some way to ensure obedience to a subtype relation.

To supply Smalltalk-80 programs with a notion of nominal type, one can annotate one's programs with this information. The Smalltalk-80 programs in Goldberg and Robson's book [GR83] already follow a convention of putting type information into variable names to aid understanding.

There are two ways to force expressions to obey a subtype relation: dynamic or static checking. Notice that we cannot rely on the dynamic type-checking of Smalltalk-80 to ensure obedience, because Smalltalk-80 only checks that an instance operation invoked on an object q is defined by q's class.

Dynamic checking could use the type information available at run-time in Smalltalk-80 programs. One would place code in all operations to check that all the operation's arguments have a type that is a subtype of their nominal type. (Smalltalk-80 itself checks the first or "controlling" argument, so no checking on the first argument is needed.) We say that a Smalltalk-80 program with such dynamic type checks is obedient if these checks never detect an instance of some type other than a subtype. Of course, there is no general algorithm for deciding when a Smalltalk-80 program is obedient.

Another way to ensure obedience would be to do static type-checking using the nominal type information added to programs as annotations. It should be easy to adapt the NOAL type system to Smalltalk-80, which would allow us to do some type checking by hand, or to write a tool that used program annotations to do static type checking.

In reasoning about Smalltalk-80 programs, it is important to define satisfaction of specifications and subtype relations with respect to a set of obedient programs. Consider what happens if we take as our set of observations the denotations of all Smalltalk-80 programs. In general, operations can be applied to expressions regardless of their "type." As we noted, this may result in an "obedience" error. Consider the program

```
aPair third
```

(which is the equivalent of the NOAL expression `third(aPair)`). Suppose `aPair` denotes an instance of `IntPair`. Then the result of the above program is an obedience error, which could not happen in an environment where `aPair` denotes an `IntTriple`. So the observation defined by this program would be able to distinguish triples from pairs. With type annotations, we can decide what environments obey a subtype relation and we can use obedient programs to describe observations. If `aPair` has nominal type `IntPair`, then the above program would not be obedient. Therefore `IntTriple` would be a subtype of `IntPair` with respect to the set of obedient programs but not with respect to all Smalltalk-80 programs.

7.5 Observable Assertions and Reasoning

In this section we discuss the role that observable assertions play in our techniques for specification and verification.

In Chapter 5 we showed that we can reason about environments that obey a subtype relation using observable assertions and standard rules from logic. Furthermore, we showed that if Q is an observable assertion, then an environment that obeys a subtype relation models Q if and only if there is no way to observe that Q does not hold. So the assumptions that assertions are observable and that environments obey a subtype relation are sufficient to ensure that our method of evaluating assertions is sensible.

The soundness of our Hoare logic does not require that every assertion used in a proof be observable. Instead we only require that the preconditions of function and operation specifications and the assertions appearing in the rules [conseq] and [carry] are observable. This requirement is restrictive, but it is needed for the *proof* of the soundness of our Hoare logic given above.

However, as a practical matter one can hardly use assertions that are not observable in proofs, since the rules [conseq] and [carry] are used with great regularity.

We do not know whether the observability restrictions of our Hoare logic are also necessary for the soundness of our verification technique. The difficulty can be illustrated by the non-observable MP-assertion "`v = x`," where `v` and `x` have nominal type `Mob`. Consider an environment η defined on these identifiers in which

$$\eta(\texttt{v}) = \mathcal{M}[\![\texttt{ins(ins(new(PSchd,true),1),2)}]\!](C,\emptyset) \tag{7.60}$$

$$\eta(\texttt{x}) = \mathcal{M}[\![\texttt{ins(ins(new(PSchd,false),1),2)}]\!](C,\emptyset). \tag{7.61}$$

Since the two instances of `PSchd` have the same set of waiting integers, we have $(C,\eta) \models$

`v = x`, even though the two instances of `PSchd` are not equal. However, the imitates relation with respect to type-safe NOAL programs is so strong that we cannot see how using the assertion "`v = x`" as a precondition of a function specification would allow us to draw invalid conclusions from our Hoare logic.

So it is an open problem whether the observability requirements of our Hoare logic are necessary for soundness.

Chapter 8
Discussion

In this chapter we discuss how the results of the previous chapters may be applied to other programming languages, what extensions to our algebraic models are needed for practical applications, and future work.

8.1 Simulation in Other Programming Languages

Various languages differ in their ability to express certain observations on abstract types. This is because a program must do its work using the operations of the given types. For example, consider an object-oriented programming language that has neither an angelic choice operator nor parallelism. The result of an angelic choice between two generic invocations cannot be expressed in such a language, because there is no way to run the two generic invocations "in parallel" and choose a result of one that halts. Because NOAL includes angelic choice, it is capable of expressing such observations.

In Chapter 6 we showed that simulation relations that relate every object of a subtype to every object of a supertype determine a subtype relation (with respect to the set of all type-safe NOAL programs). Therefore, if one's programming language is less powerful than NOAL in the sense that each observation described by that language is also an observation described by a type-safe NOAL program, then simulation relations also determine subtype relations for that language. Similarly, for such a weaker language, simulation relations should be useful for Hoare-style verification as in Chapter 7.

On the other hand, if a language lacks the constructs of NOAL and built-in types like the streams of our specification language, there may be a weaker definition of simulation relations for such a language that could serve as a basis for Hoare-style verification.

8.2 Extensions Needed for Practical Applications

Two extensions of our results are needed if they are to be directly used in "real" languages such as Smalltalk-80 or Trellis/Owl. These extensions would eliminate limitations of our algebraic models of type specifications.

The most important extension would be to model mutable types and extend our definition of subtype relations and our verification techniques to handle mutation. Our algebraic models are only suited for modeling immutable types; that is, types with no time-varying state. By contrast, most object-oriented programs use mutable types.

We also did not consider parameterized abstract types. This is not a severe limitation, however, since we can describe subtype relations and reasoning for instantiations of parameterized types. Still, it would be interesting to describe the subtype relationships among parameterized types more directly and to use such relationships to derive subtype relationships on their instantiations[1].

8.3 Future Work

In this section we focus on future work in the areas of *specification*, *verification*, and language design. Before turning to these areas we briefly state some open problems.

The following is a list of various technical problems that this dissertation has left open.

- What is the relationship between the proof theory of a subtype and the proof theory of its supertypes? That is, how can one characterize subtype relations using the set of valid assertions that can be made about the objects of various abstract types? Our definition of subtype relations is model-theoretic.

- What conditions on specifications and observations guarantee that there is a largest subtype relation?

- Does every subtype relation with respect to the set of type-safe NOAL programs determines simulation relations that witness that relation? That is, does the converse of Theorem 6.3.2 hold? This appears to be a hard problem [Nip87].

[1] A related question is what kind of parameterization is necessary or useful in a language with inclusion polymorphism.

- Are the observability conditions of our Hoare logic necessary for the soundness of program verification?

8.3.1 Future Work on Specification

During design and maintenance, one sometimes adds a new abstract type to one's program. If one already knows a subtype relation on the old types, the following questions arise.

- What conditions on the specification of the new type will ensure that the old subtype relation is still a subtype relation on the new specification?
- What has to be shown to add new subtype relationships involving the new type to the old subtype relation?

Answers to these questions would allow more modular proofs of subtype relations.

Another question is whether one can factor proofs of subtype relationships by taking advantage of controlled inheritance of specifications. For example, if the specification of a type S incorporates the specification of a type T, then it should be possible to take advantage of this relationship when proving that S is a subtype of T. However, such techniques will probably depend on a proof-theoretic characterization of subtype relations.

Finally, there is work to be done in overcoming the limitations of our specification language. These limitations are discussed in Chapter 4.

8.3.2 Future Work on Verification

An important extension to our techniques for verification would be to support the verification of modules that implement abstract types (classes). There are two aspects to this problem. The first involves showing that a class meets the specification of the type it purports to implement in the presence of inclusion polymorphism. This should be straightforward, given our results on program verification, but there may be some subtleties that we do not yet understand. The second aspect is how to factor the proof of correctness for a subclass to take advantage of the proof of correctness of its superclasses.

To make our results directly applicable to existing languages, our techniques for verification need to be extended to cover imperative languages. However, it should not be difficult to adapt our Hoare-style proof system to handle imperative languages where aliasing of mutable objects is prohibited.

8.3.3 Future Work on Language Design

In this subsection we discuss some directions for future work in language design.

One long-range project would be to design an object-oriented programming language that would support inclusion polymorphism, subtyping, inheritance, and program verification. Such a language should have a type system that can ensure obedience to a subtype relation. However, it is too early to tell what other features a language would need to support program verification. For example, we do not know what features of an inheritance mechanism help or hinder verification.

A more modest language design project would be to solve the name-clash (or interface control) problem for languages with generic invocation mechanisms [LL85]. In a language with generic invocation, each object's instance operations form a behavioral interface that is analogous to the behavioral interface of an abstract type. However, in all languages with generic invocation mechanisms that we know, there is no way to change an object's interface. Therefore each object presents the same interface to all parts of a program (except for the class that implements the object's behavior). It can be difficult and costly to combine independently designed program parts that assume that the same instance operation means different things. Furthermore, subtyping depends on object interfaces. For example, a type `Int3` that behaves like `IntTriple`, but has instance operations named `fst` and `snd`, will not be a subtype of `IntPair` even though instances of type `Int3` otherwise behave like instances of `IntPair`. A mechanism to mediate between independently designed abstractions with fixed interfaces is a feature of several languages without generic invocation mechanisms (e.g., Argus [LDH*87] and Ada [Ada83]), where one can change the interface of a type parameter. In a language with a generic invocation mechanism, one wants to be able to change the interfaces of *objects*. The ability to change object interfaces could also be exploited to provide access control for objects [JL76] [JL78].

Chapter 9

Summary and Conclusions

In this chapter we offer a high-level summary of our results and their significance as well as some conclusions about programming and programming language design.

9.1 Summary of Results

The two main results in this dissertation are a new definition of subtype relations and new techniques for the specification and verification of object-oriented programs that exploit inclusion polymorphism.

We have given a precise definition of subtype relations. This definition embodies the intuition that each instance of a subtype imitates some instance of that type's supertypes. So programs can manipulate instances of a subtype as if they were instances of that type's supertypes without surprising results.

The most important property of our definition of subtype relations is that it allows *abstract* types to be compared, based on their specifications. Most other work on subtyping only describes subtype relationships for a fixed set of built-in types (e.g., [Car84]). Our definition also allows incompletely specified and nondeterministic types to be compared, so it is more widely applicable than Bruce and Wegner's definition [BW87].

We allow nondeterminism in algebraic models and in observations because many interesting abstract types are nondeterministic and because nondeterminism allows one more freedom to leave implementation decisions open.

To deal with incomplete specifications we have taken a loose view of the semantics of a specification; that is, the semantics of a specification is a set of algebraic models instead of a single model such as the initial or final model. A loose view of specifications enables the definition of subtype relations to be applied even to specifications for which no single model captures all the desired behaviors.

We have described a way to evaluate assertions that are tailored to nominal types in programs that exploit inclusion polymorphism. This method uses the imitates relation and nominal environments. It allows us to write specifications for functions and programs that exploit inclusion polymorphism. We have shown that this method is sensible for reasoning about environments that obey a subtype relation.

Finally, we have presented a logic for Hoare-style verification of NOAL programs. This logic is novel in that it allows one to translate assertions tailored to a subtype into assertions that are tailored to a supertype. Symbolic verification of NOAL programs is done as follows. One must first find simulation relations that witness the presumed subtype relation of the abstract type specification referenced by one's program specification. These simulation relations are then used to derive axioms for translating assertions. One then uses the proof rules to prove that the purported implementation meets the specification, in the usual way. If these conditions (and a few minor technical ones) are met, then the purported implementation satisfies the program specification. This is, as far as we know, the first systematic technique for the verification of programs that exploit inclusion polymorphism.

9.2 Conclusions for Programmers

In this section we describe the significance of our results and some lessons for programmers who work with object-oriented programming languages that have generic invocation mechanisms.

Our work gives programmers a new tool: subtype relations. Subtype relationships are similar to satisfaction relationships among abstract types; the difference is largely that the syntax of class operations does not matter for a subtype relationship. Subtype relations are useful during program design, where they can help track the evolution of abstractions, limit the effects of specification changes, and group and classify related types [Lis88]. In a system like Smalltalk-80 where classes are also objects, subtype relationships among the types of classes can also be used in similar ways. We have shown how subtype relations can be used to write polymorphic specifications and to support careful reasoning.

Perhaps the most important lesson for programmers is the most basic one: subtype relationships are based on observable behavior and they have nothing to do with how a type is implemented [Sny86a]. That is, a subtype is *not* a subclass. While it is usefu' to record inheritance relationships among implementations in a subclass relation, one should organize abstract types by a subtype relation. This distinction between subclasses and subtypes, when properly understood, can be a powerful tool for separation of concerns. Subtype relations allow one to reason abstractly about instances of abstract types. Subclass relations allow one to reason about how instances are implemented.

The distinction between subtypes and subclasses is not just academic. If one passes an argument whose type is not a subtype of the expected formal argument type to a procedure, one has no guarantee that the procedure will act as desired. If one uses an instance of a subclass where instances of a superclass are expected, then one's programs may behave in unexpected ways. To prevent such problems one should ensure that each expression denotes an object whose type is a subtype of the expression's nominal type. If one programs in a statically type-checked language like Trellis/Owl, then the type system can check this second property automatically, once it has been told about a subtype relation.

An understanding of subtype relations also gives programmers a strategy for testing modules that exploit inclusion polymorphism. That is, one should concentrate on tests where the argument types are the same as the nominal types of the module's formal arguments. If there are problems in the module that can be uncovered by testing, this strategy can uncover them.

9.3 Conclusions for Language Designers

In this section we discuss some lessons for designers of new programming languages with generic invocation mechanisms.

9.3.1 Languages Should Have Declared Subtype Relations

If one is designing a type system for an object-oriented programming language with a generic invocation mechanism, then subtype relations should be a part of that type system. (*Otherwise programs will not be able to exploit inclusion polymorphism.*) Perhaps the most important lesson that emerges from our work for language designers is to make the programmer declare the subtype relation for abstract types.

The reason this lesson is so important is that the programming language cannot, in general, find a nontrivial subtype relation on the types of a program. Most programming languages are not designed to include behavioral specifications as part of programs. Each module is a specification of that module's behavior, but it is not the specification that the programmer worked from during design (and verification). Even if the program text included a behavioral specification, the problem of finding a nontrivial subtype relation on the types of a specification is undecidable in general. It seems more straightforward to let the programmer declare a subtype relation. Finally, a programmer may wish to work in a subset of a full language, and thus may only be concerned with subtype relations with respect to that subset of programs.

On the other hand, some mild restrictions on what subtype relations can be declared are probably unavoidable if one wants to use a static type system to ensure that every environment obeys the declared subtype relation. For example, to ensure obedience to a subtype relation both the NOAL and Trellis/Owl type systems require that the declared subtype relation be reflexive and transitive and that each instance operation of a supertype is also an instance operation of each of that type's subtypes (with an appropriate signature).

9.3.2 TypeOf Operators Cause Problems for Reasoning

Another lesson for language designers is that operators that tell the type of an object cause problems for reasoning and should thus be avoided. This is a new twist on an old lesson: if one wants to reason about abstract types based on their specifications, then one's language should only allow objects to be observed by invoking their instance operations.

A `typeOf` operator returns the type of an object as a string. For example, the program `typeOf(x)` will give different results in environments where `x` denotes objects of different types. Such an operator destroys subtyping. It is easy to show that a subtype relation with respect to the set of all programs that use `typeOf` cannot relate different types. So we cannot directly use our methods to reason about an implementation that uses a `typeOf` operator.

Appendix A

Summary of Notation

In this appendix we summarize the notation used in earlier chapters. We also give the signatures of the questions posed in earlier chapters, where these questions usually take the form of definitions.

Table A.1 lists some primitive domains, which are just sets. Algebras are heterogeneous (i.e., sorted), so one should think of Object as the disjoint union of several sets (one for each sort). In this appendix the carrier sets of various sorts are not distinguished for the sake of simplicity.

As in Table A.1, phrases are often abbreviated. For example, "GenOpSym" should be read as "generic operation symbol."

The syntax of the terms in our specification language is given in Figure 2.4 on page 31. The syntax of NOAL expressions is given in Figure 3.1 on page 46.

The following tables are organized by topic, which is roughly by chapter, except that the syntax and semantics of specifications and programs are treated separately. For each

Table A.1: Primitive Domains

Notation for Members	Name	description
o, q, r $\in$	Object	instances
$\mathtt{S}$, $\mathtt{T}$ $\in$	Type	type symbols
S, T $\in$	Sort	sort symbols
$\mathbf{g}_{\vec{\mathtt{S}}\to\mathtt{T}}$ $\in$	OpSymbol	operation symbols (of algebras)
$\mathbf{x}$, $\mathbf{y}$, $\mathbf{z}$ $\in$	Identifier	identifiers
$\mathbf{f}$ $\in$	FunIdent	function identifiers
$\mathbf{g}$ $\in$	GenOpSym	generic operation symbols
f $\in$	TrtFunSym	trait function symbols
true, *false* $\in$	Bool	the booleans
P, Q, R, $\in$	Term	logical formulas
E $\in$	Expr	programming language expressions

Table A.2: Algebras and Related Concepts

$o, q, r \in$	CarrierSet	$=$ Object
$o, q, r \in$	TypeCarrier	$=$ Object
$o, q, r \in$	PossRes	$=$ TypeCarrier
$Q, R \in$	SetOfPossRes	$=$ {PossRes}
$A_{\mathfrak{f}_{\mathbf{S}\to\mathbf{T}}} \in$	Operation	$=$ TypeCarrier* $\to$ SetOfPossRes
$A_{\mathfrak{f}} \in$	TraitFun	$=$ Object* $\to$ Object
$A_{OPS} \in$	SetOfOperation	$=$ {Operation}
$A_{TFUNS} \in$	SetOfTrtFun	$=$ {TraitFun}
$A, B, C \in$	Algebra	$=$ (CarrierSet, TypeCarrier, SetOfTrtFun, SetOfOperation)
$SPEC \in$	SpecSemantics	$=$ {Algebra}

Table A.3: Questions for Algebras

has type? : Object, Algebra, Type $\to$ Bool

topic there are one or two tables. One table is organized like Table A.1 and describes the domains related to that topic. The second table lists the significant questions (i.e., definitions) related to that topic.

The following conventions are used to describe domains. Each entry has the form $d \in D = E'$ meaning that d is the typical notation for an element of the domain D, which is defined by E'. For example

$$\eta \in \text{Env} = \text{TypedIdent} \to \text{Object}$$

means that η is used to denote environments, which are mappings from typed identifiers to objects. The following notations are used in describing domains. The notation $\{D\}$ means a nonempty set of elements from the domain named D. The notation D^* stands for all finite tuples of zero or more Ds. The notation $D_1 \to D_2$ denotes the set of functions from a subset of D_1 to D_2. The notation (D_1, D_2) stands for the set of all pairs whose first element is from D_1 and whose second element is from D_2.

Table A.2 describes algebras and some related operations from Chapter 2,

Table A.3 describes questions for algebras.

Table A.4 describes the concepts used to describe the syntax and semantics of the type specification language of Chapter 2. Also included are concepts used to describe the syntax of algebras. The structure of Larch traits is not further described. The ab-

Table A.4: Type Specifications and Related Concepts

$SORTS \in$	SetOfSorts	$= \{\text{Sort}\}$
$TYPES \in$	SetOfTypes	$= \{\text{Type}\}$
$V \in$	VisibleTypes	$= \{\text{Type}\}$
$TFUNS \in$	SetOfTrtFunSym	$= \{\text{TrtFunSym}\}$
$OPS \in$	SetOfOpSym	$= \{\text{OpSymbol}\}$
$\Sigma \in$	Signature	$= \begin{pmatrix} \text{SetOfSorts, SetOfTypes, VisibleTypes,} \\ \text{SetOfTrtFunSym, SetOfOpSym} \end{pmatrix}$
$\mathrm{T} \in$	Trait	$=$
$P, Q, R \in$	Assertion	= Term
$P, R \in$	Requires	= Assertion
$Q \in$	Effect	= Assertion
$\mathbf{g} \in$	OpSpec	= (GenOpSym,NomSig,Requires,Effect)
$\mathtt{T} \in$	OperationsSpec	$= \{\text{OpSpec}\}$
$\leq \in$	BinRelTyp	= Type, Type $\rightarrow$ Bool
$SPEC \in$	TypeSpec	= (SetOfTypes, BinRelTyp, Trait, OperationsSpec)
$A \in$	FunStruct	= (CarrierSet, SetOfAbsFun)
$A_{(\Sigma)} \in$	ReductOf	= Algebra, Signature $\rightarrow$ Algebra
$SIG \in$	SigOfSpec	= TypeSpec $\rightarrow$ Signature
$SIG \in$	SigOfAlg	= Algebra $\rightarrow$ Signature
$\overline{\eta} \in$	ExtendedEnv	= Term $\rightarrow$ Object

Table A.5: Questions for Type Specifications

satisfies? :	Operation, OpSpec $\rightarrow$ Bool
satisfies? :	Algebra, TypeSpec $\rightarrow$ Bool

breviation "FunStruct" stands for the "functional structure" of an algebra. The domain "BinRelTyp" should be read as "binary relations on types" or "presumed subtype relations." The notation $A_{(\Sigma)}$, where Σ is a signature, means the Σ-reduct of the algebra A. The notation $\overline{\eta}$ denotes the extension of the environment η to a mapping from terms to the elements of the carrier set of the algebra in the range of η (see Chapter 2).

Table A.5 describes the definitions satisfaction from in Chapter 2.

Table A.6 describes the concepts used to define observations in Chapter 3.

Table A.6: Observations and Related Concepts

$\mathbf{x}: \mathtt{T} \in$	TypedIdent	= (Identifier, Type)
$\eta \in$	Env	= TypedIdent $\rightarrow$ Object
$P \in$	Observation	= Algebra, Env $\rightarrow$ SetOfPossRes
$OBS \in$	SetOfObs	$= \{\text{Observation}\}$

Table A.7: Programming Language Concepts

$X, Y, Z \in$	SetOfIdent	= {TypedIdent}
$\mathcal{M} \in$	Denotation	= Expr → Observation
$Generic \in$	GIMap	= GenOpSym, TypeCarrier*, Algebra → OpSymbol
$GOP \in$	SetOfGenOp	= {GenOpSym}
$\vec{\mathtt{S}} \to \mathtt{T} \in$	NomSig	= (Type*, Type)
$\{\vec{\mathtt{S}} \to \mathtt{T}\} \in$	SetOfNomSig	= {NomSig}
$NomSig \in$	NomSigMap	= GenOpSym → SetOfNomSig
$H, X \in$	TypeAssumptions	= {TypedIdent}
$\sqsubseteq \in$	DomainOrder	= Object, Object → Bool
$\sqsubseteq_E \in$	DomOrdForSets	= SetOfPossRes, SetOfPossRes → Bool
$\overline{Q} \in$	ClosureOf	= SetOfPossRes → SetOfPossRes

Table A.8: Questions for Programming Language

has nominal type?	: Expr, NomSigMap, BinRelTyp, Type → Bool
safe?	: BinRelTyp, NomSigMap → Bool
obeys?	: Algebra, Env, BinRelTyp → Bool
monotonic?	: Operation → Bool
strongly monotonic?	: Operation → Bool
continuous?	: Operation → Bool

Table A.7 describes the concepts used in Chapter 3 and Appendix C to give semantics to NOAL programs and to describe the NOAL type system. The operator that takes the closure of a set is written as an overbar; that is, the closure of a set Q is written $\overline{Q}$.

Table A.8 describes the major definitions of Chapter 3 and Appendix C.

Table A.9 describes the concepts for function specifications from Chapter 4, and table A.10 describes the major definitions.

Table A.11 describes subtype relations from Chapter 5.

Table A.12 describes typed families of relations, abbreviated "FamilyOfRel," from Chapter 6. Table A.13 describes the definition of simulation relations from Chapter 6.

Table A.14 describes the concept of a Hoare-triple from Chapter 7 and table A.15 describes related definitions.

Table A.9: Function Specification Concepts

$\mathtt{f} \in$	FunSpec	= (FunIdent,NomSig,TypeSpec,Requires,Effect)
$c_Q \in$	CharObserv	= Observation

Table A.10: Questions for Function Specifications

nominal?	: Algebra, Env $\rightarrow$ Bool
imitates?	: Algebra, Env, Algebra, Env, SetOfObs $\rightarrow$ Bool
models?	: Algebra, Env, Assertion $\rightarrow$ Bool
proper?	: Env $\rightarrow$ Bool
satisfies?	: FunSpec, Operation $\rightarrow$ Bool
observable?	: Assertion, SetOfObs $\rightarrow$ Bool

Table A.11: Subtype Relations

subtype relation?	: SpecSemantics, BinRelTyp, SetOfObs $\rightarrow$ Bool

Table A.12: Typed Families Of Relations

$\mathcal{R} \in$ FamilyOfRel = { TypeCarrier, TypeCarrier $\rightarrow$ Bool }

Table A.13: Questions for Simulation

bistrict?	: Algebra, Algebra, FamilyOfRel $\rightarrow$ Bool
V-identical?	: Algebra, Algebra, FamilyOfRel $\rightarrow$ Bool
homomorphic rel?	: Algebra, Algebra, FamilyOfRel, NomSigMap, BinRelTyp $\rightarrow$ Bool
simulation rel?	: Algebra, Algebra, FamilyOfRel, NomSigMap, BinRelTyp $\rightarrow$ Bool
universal model?	: Algebra, SpecSemantics, SetOfObs $\rightarrow$ Bool

Table A.14: Verification Concepts

$P \in$ PreCond	= Assertion
$Q \in$ PostCond	= Assertion
P {y $\leftarrow$ E} $Q \in$ HoareTriple	= (PreCond,Identifier,Expression,PostCond)

Table A.15: Questions for Verification

models?	: Algebra, Env, HoareTriple $\rightarrow$ Bool
valid?	: SpecSemantics, BinRelTyp, HoareTriple $\rightarrow$ Bool

Appendix B

Visible Types and Streams

In this appendix we describe the models of the visible types fixed by our type specification language. These types are `Bool`, `Int` and two corresponding stream types: `BoolStream` and `IntStream`.

The algebra for the type `Bool` is found in Figure 2.1 on page 26.

Our algebra for the type `Int` is found in Figure B.1.

The types `IntStream` and `BoolStream` are used to model output. The `cons` operation of each type is lazy; that is, `cons` is not strict in its second argument. In Figure B.2 on page 188 we give an algebraic model of `IntStream`. The model of `BoolStream` is similar and can be obtained by replacing `Bool` for `Int` throughout.

The carrier set of `IntStream` is defined using the operator *Stream* [Bro86], defined as

$$Stream(I) \stackrel{\text{def}}{=} \{I^* \cup (I^* \times \{\bot\}) \cup I^\infty\}, \tag{B.1}$$

where

- I^* denotes the set of *finite streams*, which are finite sequences of elements of I, such as the empty stream $\langle\rangle$ and $\langle i_1, i_2, i_3\rangle$,
- $I^* \times \{\bot\}$ denotes the set of *partial streams*, which are finite sequences ending in $\bot$, such as $\langle i_1, i_2, i_3, \bot\rangle$ and the totally undefined stream $\bot = \langle\bot\rangle$, and
- I^∞ denotes the set of *infinite streams*, such as $\langle i_1, i_2, i_3, \ldots\rangle$.

The definition of the $\texttt{rest}_{\texttt{IntStream}\rightarrow\texttt{IntStream}}$ operation also needs some explanation. The `rest` operation is strict, as Figure B.2 shows, since all trait functions are strict. Furthermore, one should think of the `rest` operation as requiring that its argument stream

Figure B.1: Model of the visible type `Int`.

Carrier sets

$$B_{\texttt{Int}} \stackrel{\text{def}}{=} \{\bot, 0, 1, -1, 2, -2, \ldots\}$$
$$B_{\texttt{IntClass}} \stackrel{\text{def}}{=} \{\bot, \mathit{Int}\}$$

Trait Functions

$$B_{\text{Int}}() \stackrel{\text{def}}{=} \mathit{Int}$$
$$B_0() \stackrel{\text{def}}{=} 0$$
$$B_1() \stackrel{\text{def}}{=} 1$$
$$B_{\#+\#}(i,j) \stackrel{\text{def}}{=} i + j$$
$$B_{\#-\#}(i,j) \stackrel{\text{def}}{=} i - j$$
$$B_{-\#}(i) \stackrel{\text{def}}{=} -i$$
$$B_{\#*\#}(i,j) \stackrel{\text{def}}{=} i \cdot j$$
$$B_{\#.\text{eq}\#}(i,j) \stackrel{\text{def}}{=} \begin{cases} \mathit{true} & \text{if } i = j \\ \mathit{false} & \text{otherwise} \end{cases}$$
$$B_{\#<\#}(i,j) \stackrel{\text{def}}{=} \begin{cases} \mathit{true} & \text{if } i < j \\ \mathit{false} & \text{otherwise} \end{cases}$$
$$B_{\#\leq\#}(i,j) \stackrel{\text{def}}{=} B_{\#|\#}(B_{\#<\#}(i,j), B_{\#.\text{eq}\#}(i,j))$$
$$B_{\#>\#}(i,j) \stackrel{\text{def}}{=} B_{\#<\#}(j,i)$$
$$B_{\#\geq\#}(i,j) \stackrel{\text{def}}{=} B_{\#\leq\#}(j,i)$$

Operations

$$B_{\texttt{Int}_{\to\texttt{IntClass}}}() \stackrel{\text{def}}{=} \{\mathit{Int}\}$$
$$B_{\texttt{one}_{\texttt{IntClass}\to\texttt{Int}}}(\mathit{Int}) \stackrel{\text{def}}{=} \{1\}$$
$$B_{\texttt{add}_{\texttt{Int},\texttt{Int}\to\texttt{Int}}}(i,j) \stackrel{\text{def}}{=} \{B_{\#+\#}(i,j)\}$$
$$B_{\texttt{neg}_{\texttt{Int}\to\texttt{Int}}}(i) \stackrel{\text{def}}{=} \{B_{-\#}(i)\}$$
$$B_{\texttt{sub}_{\texttt{Int},\texttt{Int}\to\texttt{Int}}}(i,j) \stackrel{\text{def}}{=} \{B_{\#-\#}(i,j)\}$$
$$B_{\texttt{mul}_{\texttt{Int},\texttt{Int}\to\texttt{Int}}}(i,j) \stackrel{\text{def}}{=} \{B_{\#*\#}(i,j)\}$$
$$B_{\texttt{equal?}_{\texttt{Int},\texttt{Int}\to\texttt{Bool}}}(i,j) \stackrel{\text{def}}{=} \{B_{\#.\text{eq}\#}(i,j)\}$$
$$B_{\texttt{lt?}_{\texttt{Int},\texttt{Int}\to\texttt{Bool}}}(i,j) \stackrel{\text{def}}{=} \{B_{\#<\#}(i,j)\}$$

not be empty, since the set of possible results of invoking **rest** on an empty stream is the entire carrier set of **IntStream**. Finally, note that the $\mathtt{cons}_{\mathtt{IntStream},\mathtt{Int}\rightarrow\mathtt{IntStream}}$ operation is not strict in its stream argument, as this is how partial streams are constructed.

Figure B.2: Model of the visible type IntStream.

Carrier sets

$$
\begin{aligned}
B_{\texttt{IntStream}} &\overset{\text{def}}{=} \mathit{Stream}(\{0,1,-1,\ldots,\}) \\
B_{\texttt{IntClass}} &\overset{\text{def}}{=} \{\perp, \mathit{IntStream}\}
\end{aligned}
$$

Trait Functions

$$
\begin{aligned}
B_{\text{IntStream}}() &\overset{\text{def}}{=} \mathit{IntStream} \\
B_{\text{empty}}() &\overset{\text{def}}{=} \langle\rangle \\
B_{\text{cons}}(s,i) &\overset{\text{def}}{=} \begin{cases} \langle i\rangle & \text{if } s = \langle\rangle \\ \langle i, i_1, \ldots\rangle & \text{if } s = \langle i_1, \ldots\rangle \end{cases} \\
B_{\text{first}}(s) &\overset{\text{def}}{=} \begin{cases} 0 & \text{if } s = \langle\rangle \\ i_1 & \text{if } s = \langle i_1, \ldots\rangle \end{cases} \\
B_{\text{rest}}(s) &\overset{\text{def}}{=} \begin{cases} \langle\rangle & \text{if } s = \langle\rangle \\ \langle\rangle & \text{if } s = \langle i, \perp\rangle \\ \langle i_1, \ldots\rangle & \text{if } s = \langle i, i_1, \ldots\rangle \end{cases} \\
B_{\text{isEmpty?}}(s) &\overset{\text{def}}{=} \begin{cases} \mathit{true} & \text{if } s = \langle\rangle \\ \mathit{false} & \text{if } s = \langle i_1, \ldots\rangle \end{cases}
\end{aligned}
$$

Operations

$$
\begin{aligned}
B_{\texttt{IntStream}\to\texttt{IntClass}}() &\overset{\text{def}}{=} \{\mathit{IntStream}\} \\
B_{\texttt{empty}_{\texttt{IntStreamClass}\to\texttt{IntStream}}}(\mathit{IntStream}) &\overset{\text{def}}{=} \{\langle\rangle\} \\
B_{\texttt{undefined}_{\texttt{IntStreamClass}\to\texttt{IntStream}}}(\mathit{IntStream}) &\overset{\text{def}}{=} \{\perp\} \\
B_{\texttt{first}_{\texttt{IntStream}\to\texttt{Int}}}(s) &\overset{\text{def}}{=} \begin{cases} \{B_{\text{first}}(s)\} & \text{if } s \neq \langle\rangle \\ \{\perp, 0, 1, -1, \ldots\} & \text{if } s = \langle\rangle \end{cases} \\
B_{\texttt{rest}_{\texttt{IntStream}\to\texttt{IntStream}}}(s) &\overset{\text{def}}{=} \begin{cases} \{B_{\text{rest}}(s)\} & \text{if } s \neq \langle\rangle,\ s \neq \langle i, \perp\rangle \\ \{\perp\} & \text{if } s = \langle i, \perp\rangle \\ B_{\texttt{IntStream}} & \text{if } s = \langle\rangle \end{cases} \\
B_{\texttt{cons}_{\texttt{IntStream},\texttt{Int}\to\texttt{IntStream}}}(s,i) &\overset{\text{def}}{=} \{B_{\text{cons}}(s,i)\} \\
B_{\texttt{cons}_{\texttt{IntStream},\texttt{Int}\to\texttt{IntStream}}}(\perp,i) &\overset{\text{def}}{=} \{\langle i, \perp\rangle\} \\
B_{\texttt{isEmpty?}_{\texttt{IntStream}\to\texttt{Bool}}}(s) &\overset{\text{def}}{=} \{B_{\text{isEmpty?}}(s)\}
\end{aligned}
$$

Appendix C

Recursively-Defined NOAL Functions

In this appendix we describe the semantics of systems of recursively-defined functions in NOAL and we prove the substitution property for NOAL functions.

C.1 Semantics of NOAL Functions

The semantics of NOAL functions are discussed informally in Chapter 3. In this section we give a formal semantics for systems of mutually recursive NOAL functions. Our semantics follows Broy's discussion of the semantics of AMPL [Bro86, Page 20]. We also describe how the carrier sets of an algebra are viewed as domains and an assumption about the domain ordering on the carrier sets of algebras that can be observed by NOAL programs.

Throughout this section we fix a signature $\Sigma = (\mathit{SORTS}, \mathit{TYPES}, V, \mathit{TFUNS}, \mathit{OPS})$ and a Σ-algebra A.

C.1.1 Domains and Domain Orderings

Recall that $\perp$, which represents nontermination and errors, is an element of the carrier set of each type in an algebra. To define the semantics of NOAL (in particular the semantics of recursive function definitions), we use a partial order $\sqsubseteq$ on each type's carrier set that makes that carrier set a domain, that is a pointed complete partial order.

The following definition of a pointed complete partial order is taken from [Sch86, Page 111]. For a partially ordered set D, a subset Q of D is a *chain* if it is nonempty and for all $q_1, q_2 \in Q$, either $q_1 \sqsubseteq q_2$ or $q_2 \sqsubseteq q_1$. A *complete partial order* is a set D with a partial order $\sqsubseteq$, such that every chain in D has a least upper bound in D. The *least upper bound* of a chain $Q \subseteq D$, written $\mathrm{lub}(Q)$, is the smallest element of D that is at least as large as every element of Q. A *pointed complete partial order* is a complete partial order that has a least element, $\perp$.

From now on we will simply refer to pointed complete partial orders as *domains*. We are primarily interested in flat domains, since the semantics for recursive functions that we use assumes that each carrier set, except for the carrier sets of the stream types, is a flat domain [Bro86, Page 7].

Definition C.1.1 (flat domain). A domain is *flat* if and only if for all elements q and r, $q \sqsubseteq r$ if and only if $q = r$ or $q = \perp$.

As usual, we write $q \sqsubset r$ if $q \sqsubseteq r$ and $q \neq r$. Therefore, in a flat domain, $q \sqsubset r$ if and only if $q = \perp$.

Our assumptions about the partial order $\sqsubseteq$ on a carrier set are as follows. Let Σ be a signature and let A be a Σ-algebra. Recall that the semantics of the visible types is fixed by convention; that is, the same reduct is used in all algebras for the visible types. We assume that `Bool` is a visible type with proper elements *true* and *false*; these are needed to define `if` expressions. We assume that for each visible type v, the carrier set of v comes equipped with a partial order $\sqsubseteq$ (defined by convention) that makes A_v a domain with $\perp$ as its least element. We further assume that the carrier set of each visible type except `BoolStream` and `IntStream` is a flat domain. For a non-visible type `T`, we define $\sqsubseteq$ so that the carrier set of `T` is a flat domain.

For example, our convention for $\sqsubseteq$ on the carrier sets of the visible types of the algebraic models of our specification language is as follows. The carrier sets of `Bool` and `Int` are flat domains. The carrier sets of `BoolStream` and `IntStream` are such that if A is an algebra and $q, r \in A_{\texttt{BoolStream}}$, then $q \sqsubseteq r$ if and only if either $q = r$ or q is a partial stream whose proper elements are a prefix of r [Bro86, Section 2.1].

We also regard the carrier set of A itself as a domain formed by the union of all its carrier sets. That is, $q \sqsubseteq r$ in A if and only if q and r are in the same carrier set and $q \sqsubseteq r$. (Recall that $\perp$ is in each type's carrier set.)

For the domain ordering on an algebra to be useful, it must say something about the operations of the algebra. In particular, the operations of the algebra must be monotonic and continuous. To define these terms, we first extend the partial order $\sqsubseteq$ to tuples as follows: $\vec{q} \sqsubseteq \vec{r}$ if and only if for all i, $q_i \sqsubseteq r_i$. Since operations return a set of possible results, we also define an ordering $\sqsubseteq_E$ on sets of possible results. We write $Q \sqsubseteq_E R$ if for each $q \in Q$ there is some $r \in R$ such that $q \sqsubseteq r$ [Bro86, Page 13].

Definition C.1.2 (monotonic). An operation g is *monotonic* if and only if for all $\vec{q_1}$, $\vec{q_2}$, if $\vec{q_1} \sqsubseteq \vec{q_2}$, then $g(\vec{q_1}) \sqsubseteq_E g(\vec{q_2})$.

That is, g is monotonic if whenever $\vec{q_1} \sqsubseteq \vec{q_2}$ and $r_1 \in g(\vec{q_1})$, then there is some $r_2 \in g(\vec{q_2})$ such that $r_1 \sqsubseteq r_2$.

To define continuous operations, we view chains as sequences. A *sequence in* $\sqsubseteq$ is a nonempty set $Q = \{q_i \mid i \in I\}$ indexed by some well-ordered set I (whose elements are ordered by $\leq$) with the property that, if $i \leq j$, then $q_i \sqsubseteq q_j$. A *well-ordered set* is a totally-ordered set such that every non-empty subset has a least element [Gra79, Page12]. The elements of a sequence form a chain and conversely the elements of a chain can be placed in a sequence. We use one concept or the other as is convenient.

Definition C.1.3 (continuous). A monotonic operation g is *continuous* if and only if for every sequence in $\sqsubseteq$, $Q = \{q_i\}$, whenever $R = \{r_i\}$ is a sequence in $\sqsubseteq$ indexed by the same set as Q such that for all indexes i, $r_i \in g(\vec{q_i})$, then $\text{lub}(R) \in g(\text{lub}(Q))$.

Because we require the operations of an algebra to be continuous, our assumption that the carrier sets of all types except `IntStream` and `BoolStream` are flat domains is restrictive. That is, there are some abstract types whose carrier sets cannot be consid flat domains if their operations are to be monotonic and continuous. For example, the carrier set of `IntStream` cannot be a flat domain, since then the **cons** operation would not be monotonic.

So that we may assign denotations to systems of mutually recursive function definitions, we also assume that each set of possible results of an algebra's operations is closed with respect to the algebra's domain ordering $\sqsubseteq$. A set of values is *closed* if and only if for every chain $Q \subseteq D$, its least upper bound, $\text{lub}(Q)$, is also in D. This assumption ensures that the set of possible results of each NOAL expression is closed and thus accords with the principle of finite observability [Bro86].

C.1.2 Semantics of Recursive Functions in NOAL

In this subsection we give the semantics of systems of mutually recursive NOAL functions.

We begin by defining the semantics of systems that do not use angelic choice. Following Broy we obtain approximations by eliminating erratic choice operators ($\Box$) and textually expanding recursive calls. Erratic choices are turned into different expansions.

The notation $\mathcal{F}(A)[\![\mathtt{f}]\!]$ stands for the denotation of a function definition named **f** in an algebra A. The denotation of a system of function definitions does not depend on the surrounding environment, because in the body of a recursively defined NOAL function, there can be no free identifiers or function identifiers, besides those of the other recursively defined functions and the function's formal arguments.

Fix an algebra A. Let

$$\begin{array}{l} \mathbf{fun}\ \mathbf{f}_1(\vec{\mathbf{x_1}} : \vec{\mathrm{S_1}}) : \mathrm{T}_1 = E_1; \\ \vdots \\ \mathbf{fun}\ \mathbf{f}_m(\vec{\mathbf{x}_m} : \vec{\mathrm{S}_m}) : \mathrm{T}_m = E_m \end{array}$$

be a mutually recursive system of NOAL function definitions, where the angelic choice operator ($\bigtriangledown$) does not occur in the E_j.

When eliminating erratic choice operators one makes choices of what expressions to execute; each such choice is called a deterministic descendant. An expression E'' is a *deterministic descendant of an expression E' if E'' does not contain the erratic choice* operator ($\Box$) and can be obtained from E' by replacing subexpressions of the form $\gamma_1 \Box \gamma_2$ with either γ_1 or γ_2.

A family $D_{(j,i)}$ of expressions is called a *choice family* for the system of $\mathbf{f}_j$ if for each j, $D_{(j,0)}$ is a deterministic descendant of E_j, and $D_{(j,i+1)}$ is a deterministic descendant of $D_{(j,i)}$ with $(\mathbf{fun}(\vec{\mathbf{x}_k} : \vec{\mathrm{S}_k})E_k)$ substituted for $\mathbf{f}_k$ for each k. The expression $D_{(j,i+1)}$ differs from $D_{(j,i)}$ in that one more recursion is unrolled, thus $D_{(j,i+1)}$ is a better approximation to one computation of $\mathbf{f}_j$ than $D_{(j,i)}$.

For example, consider a system with one recursively defined function, where $\mathbf{f}_1$ is the function **choose** defined by

```
fun choose (x:Int): Int = (x [] choose(add(x,1))).
```

There are infinitely many choice families for this example. One choice family for **choose** is for all i, $D_{(1,i)} = \mathbf{x}$. Another choice family has $D_{(1,i)} = D_{(1,1)}$ for all $i > 1$, where

$$\begin{array}{rcl} D_{(1,0)} & = & \mathtt{choose(add(x,1))} \\ D_{(1,1)} & = & \mathtt{(fun\ (x:Int)\ x)\ (add(x,1)).} \end{array}$$

There is also a choice family that has an occurrence of **choose** in every $D_{(1,i)}$.

As usual, an everywhere-$\perp$ function is the first approximation to recursive invocations in the $D_{(j,i)}$. For each j, let $\mathtt{G}_j$ be the function abstract of the form

$$\mathtt{fun}(\vec{\mathtt{x}}_j : \vec{\mathtt{S}}_j) : \mathtt{T}_j = \mathtt{bottom}[\mathtt{T}_j].$$

To define the meaning of each recursively defined NOAL function, we take the least upper bounds of sequences of approximate results. Given a choice family, $D_{(j,i)}$, for each j let $Q_j(\vec{q}) = \langle \hat{q}_i \rangle$ be a sequence in $\sqsubseteq$, where for each i, $\hat{q}_i$ is a possible result of $\mathcal{M}[\![D_{(j,i)}[\vec{\mathtt{G}}/\vec{\mathtt{f}}]]\!](A, \eta)$ and $\eta(\vec{\mathtt{x}}_j) = \vec{q}$. As Broy notes, there are such sequences in $\sqsubseteq$ because the deterministic language constructs (and each operation of A) are monotonic and because $D_{(j,i+1)}$ is derived from $D_{(j,i)}$ by unrolling another recursion. Note that $D_{(j,i)}[\vec{\mathtt{G}}/\vec{\mathtt{f}}]$ is recursion-free. For each j, let $DD_j(\vec{q})$ denote the set of all sequences $Q_j(\vec{q})$ for all choice families.

For the **choose** example, $DD_1(0)$ would be the set consisting of the sequence $\langle \perp, \perp, \perp, \ldots \rangle$ and all sequences in $\sqsubseteq$ of the form

$$\langle \underbrace{\perp, \perp, \ldots, \perp}_{n}, n, n, n, \ldots \rangle$$

for some $n \geq 0$.

The denotation $\mathcal{F}(A)[\![\mathtt{f}_j]\!]$ is defined by

$$\mathcal{F}(A)[\![\mathtt{f}_j]\!](\vec{q}) \stackrel{\text{def}}{=} \overline{\{\mathrm{lub}(Q_j(\vec{q})) \mid Q_j(\vec{q}) \in DD_j(\vec{q})\}}. \tag{C.1}$$

That is, $\mathcal{F}(A)[\![\mathtt{f}_j]\!](\vec{q})$ is the closure of the set of all the least upper bounds of all sequences in $\sqsubseteq$ from $DD_j(\vec{q})$. The *closure* of a set Q, written $\overline{Q}$, is the smallest closed set that contains Q. Taking the closure ensures that the set of possible results is closed; it might otherwise be possible to form a sequence from the $\mathrm{lub}(Q_j(\vec{q}))$ whose least upper bound was not in the set.

For the **choose** example we determine the possible results of `choose(0)` as follows. The least upper bound of the sequence $\langle \perp, \perp, \perp, \ldots \rangle$ is $\perp$. The least upper bound of a sequences of the form $\langle \perp, \perp, \perp, \ldots, n, n, n, \ldots \rangle$ is n. So we have

$$\{\mathrm{lub}(Q_j(0)) \mid Q_j(0) \in DD(0)\} = \{\perp, 0, 1, 2, 3, \ldots\}.$$

This set is already closed in the $\sqsubseteq$ ordering (as the carrier set of **Int** is a flat domain) so it is the set of possible results.

The meaning of a system of recursive function definitions that uses angelic choice uses the meaning of a system that does not use angelic choice as a first approximation. Better approximations are obtained by using earlier approximations to evaluate recursive calls. The net effect is that each approximation uses angelic choice for deeper recursions than the previous approximation [Bro86, Page 19].

Let

$$\begin{array}{l}\texttt{fun } \mathtt{f}_1(\vec{\mathtt{x}_1} : \vec{\mathtt{S}_1}) : \mathtt{T}_1 = E_1; \\ \vdots \\ \texttt{fun } \mathtt{f}_m(\vec{\mathtt{x}_m} : \vec{\mathtt{S}_m}) : \mathtt{T}_m = E_m\end{array}$$

be a system of mutually recursive NOAL function definitions. Let $E_{(j,0)}$ be derived from E_j by replacing all occurrences of the angelic choice operator ($\bigtriangledown$) with the erratic choice operator ($[]$). Let

$$\begin{array}{l}\texttt{fun } \mathtt{g}_{(1,0)}(\vec{\mathtt{x}_1} : \vec{\mathtt{S}_1}) : \mathtt{T}_1 = E_{(1,0)}; \\ \vdots \\ \texttt{fun } \mathtt{g}_{(m,0)}(\vec{\mathtt{x}_m} : \vec{\mathtt{S}_m}) : \mathtt{T}_m = E_{(m,0)}\end{array}$$

For example, consider the function

```
fun choose2 (x:Int): Int = (x ▽ choose2(add(x,1))).
```

For this example, the system with $\bigtriangledown$ replaced by $[]$ is

```
fun g(1,0) (x:Int): Int = (x [] choose2(add(x,1))).
```

For each j, $\mathcal{F}(A)[\![\mathtt{g}_{(j,0)}]\!]$ gives meaning to recursive calls to $\mathtt{f}_j$. We call the next approximation obtained in this way $\mathcal{F}(A)[\![\mathtt{g}_{(j,1)}]\!]$. In this way a family $\mathcal{F}(A)[\![\mathtt{g}_{(j,i)}]\!]$ for each j and i is defined as follows. For each natural number i,

$$\mathcal{F}(A)[\![\mathtt{g}_{(j,i+1)}]\!](\vec{q}) \overset{\text{def}}{=} \overline{\mathcal{M}[\![E_j]\!](A,\eta)}, \tag{C.2}$$

where

$$\eta(\vec{\mathtt{x}_j}) = \vec{q} \tag{C.3}$$

and where for all k,

$$\eta(\mathtt{f}_k) = \mathcal{F}(A)[\![\vec{\mathtt{g}}_{(k,i)}]\!]. \tag{C.4}$$

That is, to find the possible results of $\mathcal{F}(A)[\![\mathbf{g}_{(j,i+1)}]\!](\vec{q})$, take the possible results of E_j, which may use angelic choice, in an environment where $\vec{q}$ is bound to the formals of $\mathbf{f}_j$ and $\mathcal{F}(A)[\![\vec{\mathbf{g}}_{(k,i)}]\!]$ is used as an approximation to $\mathbf{f}_k$, for all k. (The only free identifiers in E_j are the $\vec{\mathbf{x}}_j$, and the only free function identifiers are the $\mathbf{f}_k$.) By construction, $\mathcal{F}(A)[\![\vec{\mathbf{g}}_{(j,i)}]\!]$ does i levels of recursion using angelic choice and then reverts to erratic choice.

For the choose2 example,

$$\begin{aligned}
\mathcal{F}(A)[\![\mathbf{g}_{(1,0)}]\!](0) &= \{\bot,0,1,2,3,\ldots\} && \text{(C.5)}\\
\mathcal{F}(A)[\![\mathbf{g}_{(1,1)}]\!](0) &\overset{\text{def}}{=} \overline{\mathcal{M}[\![\mathtt{x} \bigtriangledown \mathtt{choose2(add(x,1))}]\!](A,\eta)} && \text{(C.6)}\\
&= \{0\} \cup (\{\bot,1,2,3,\ldots\} \setminus \{\bot\}) && \text{(C.7)}\\
&= \{0,1,2,3,\ldots\}. && \text{(C.8)}
\end{aligned}$$

where $\eta(\mathtt{x}) = 0$ and $\eta(\mathtt{choose2}) = \mathcal{F}(A)[\![\mathbf{g}_{(1,0)}]\!]$. By the definition of angelic choice, $\bot$ is not a possible result of the expression `x` $\bigtriangledown$ `choose2(add(x,1))` in η, because the only possible result of `x` is 0. The possible results of $\mathcal{F}(A)[\![\mathbf{g}_{(1,i)}]\!](0)$ for all $i > 1$ are also $\{0,1,2,3,\ldots\}$.

Following [Bro86, Page 19], for each j,

$$\mathcal{F}(A)[\![\mathbf{f}_j]\!](\vec{q}) \overset{\text{def}}{=} \bigcap_i \mathcal{F}(A)[\![\mathbf{g}_{(j,i)}]\!](\vec{q}). \qquad \text{(C.9)}$$

For the choose2 example:

$$\begin{aligned}
\mathcal{F}(A)[\![\mathtt{choose2}]\!](0) &\overset{\text{def}}{=} \bigcap_i \mathcal{F}(A)[\![\mathbf{g}_{(1,i)}]\!](0) && \text{(C.10)}\\
&= \{0,1,2,3,\ldots\}. && \text{(C.11)}
\end{aligned}$$

C.2 The Substitution Property for NOAL Functions

In this section we prove the substitution property for NOAL functions; that is we give the postponed proof of Lemma 6.2.2.

Because the semantics of systems of mutually recursive function definitions involve closures and least upper bounds of sequences, it is convenient to first show that simulation relations are strongly monotonic and continuous.

Definition C.2.1 (strongly monotonic). A binary relation $\ll$ between domains D_1 and D_2 is *strongly monotonic* if and only if for all $q_1, q_2 \in D_1$ and for all $r_1, r_2 \in D_2$, whenever $q_1 \sqsubseteq q_2$, $q_1 \ll r_1$, and $q_2 \ll r_2$, then $r_1 \sqsubseteq r_2$.

Figure C.1: Strong monotonicity of $\ll$.

$$\begin{tikzcd}
q_2 & \ll & r_2 & & q_2 & \ll & r_2 \\
\sqcup & & & \Longrightarrow & \sqcup & & \sqcup \\
q_1 & \ll & r_1 & & q_1 & \ll & r_1
\end{tikzcd}$$

This definition is illustrated in Figure C.1. A typed family of relations $\mathcal{R}$ is strongly monotonic if each $\mathcal{R}_T$ is a strongly monotonic relation.

The following lemma says that each simulation relation is strongly monotonic.

Lemma C.2.2. Let Σ be a signature. Let C and A be Σ-algebras such that each carrier set of a non-visible type is flat.

If $\mathcal{R}$ is a simulation relation between C and A, then $\mathcal{R}$ is strongly monotonic.

Proof: Let T be a type. Suppose $q_1 \sqsubset q_2$, $q_1 \mathcal{R}_T r_1$, and $q_2 \mathcal{R}_T r_2$. Since $q_1 \sqsubset q_2$, either $q_1 = \perp$ and q_2 is proper or both q_1 and q_2 are proper elements of some visible type with a non-flat carrier set.

If $q_1 = \perp$ and q_2 is proper, then since $\mathcal{R}_T$ is bistrict, $r_1 = \perp$ and r_2 is proper. So $r_1 \sqsubset r_2$.

If q_1 and q_2 are proper elements of a visible type then $q_1 = r_1$, and $q_2 = r_2$, because $\mathcal{R}$ is V-identical. ∎

The following lemma says that each simulation relation is continuous. A typed family of relations is continuous if it is continuous at each type.

Lemma C.2.3. Let Σ be a signature. Let A and B be Σ-algebras such that each carrier set of a non-visible type is flat.

If $\mathcal{R}$ is a simulation relation between A and B, then $\mathcal{R}$ is continuous.

Proof: Let T be a type. Let Q be a sequence in $\sqsubseteq$ of elements of A. Let R be a sequence in $\sqsubseteq$ of elements of B, indexed by the same set as Q, such that for all indexes i, $q_i \mathcal{R}_T r_i$.

If the only elements of Q are $\perp$, then the only elements of R are $\perp$, since $\mathcal{R}_T$ is bistrict; thus $\text{lub}(Q) = \perp \mathcal{R}_T \perp = \text{lub}(R)$.

Otherwise, if Q contains some proper elements, then the proper elements must all be contained in the carrier set of some type S, since they are related by $\sqsubseteq$.

If S is not a visible type, then its carrier set is flat, so the least upper bound of Q must occur in Q. Let j be an index such that $\mathrm{lub}(Q) = q_j \in Q$. Since $\mathcal{R}$ is V-identical, the proper elements of Q cannot be related by $\mathcal{R}_\mathrm{T}$ to proper elements of a visible type; so R only contains elements of a non-visible type, which is a flat domain. Since $\mathcal{R}_\mathrm{T}$ is bistrict, q_j is proper, and $q_j \mathrel{\mathcal{R}_\mathrm{T}} r_j$, it follows that r_j is proper. Since R only contains proper elements of a flat domain, $\mathrm{lub}(R) = r_j$.

If S is a visible type, then since $\mathcal{R}$ is V-identical and $\mathcal{R}_\mathrm{T}$ relates some elements of S, $\mathcal{R}_\mathrm{T}$ contains the identity on the carrier set of S. Therefore, for each i and each $q_i \in Q$, $q_i = r_i \in R$. Therefore $\mathrm{lub}(Q) = \mathrm{lub}(R)$. Since $\mathcal{R}_\mathrm{T}$ contains the identity on the carrier set of S, $\mathrm{lub}(Q) \mathrel{\mathcal{R}_\mathrm{T}} \mathrm{lub}(R)$. ∎

The following lemma says that the closures of sets related by a strongly monotonic and continuous relation are related. This lemma is needed because closures are used in the semantics of systems of recursively defined NOAL functions.

Lemma C.2.4. Let D_1 and D_2 be domains. Let $\ll$ be a strongly monotonic and continuous relation between D_1 and D_2.

If $Q \subseteq D_1$ and $R \subseteq D_2$ are such that $Q \ll R$, then $\overline{Q} \ll \overline{R}$.

Proof: Suppose $\hat{q} \in \overline{Q}$, but $\hat{q} \notin Q$. Since $\hat{q} \in \overline{Q}$, there is some sequence in $\sqsubseteq$, Q_0, consisting of elements of Q such that $\mathrm{lub}(Q_0) = \hat{q}$. Let I be the well-ordered set that indexes Q_0 and let the elements of I be ordered by $\leq$. Since $Q \ll R$, a sequence in $\sqsubseteq$ from R such that $\mathrm{lub}(Q_0)$ is related by $\ll$ to its least upper bound can be defined inductively as follows. As the basis, let i_0 be the least element of the index set I. Since $Q \ll R$, there is some $r_{i_0} \in R$ such that $q_{i_0} \ll r_{i_0}$. For the inductive step, suppose that r_k is defined for all $k \in I$ such that $k \leq j$. Let i be the least element of I such that $j < i$; then r_i can be chosen as follows. If $q_j = q_i$, let $r_i = r_j$. Otherwise, if $q_j \sqsubset q_i$, let $r_i \in R$ be such that $q_i \ll r_i$. Such an r_i exists because $Q \ll R$. Since $\ll$ is strongly monotonic, if $q_j \sqsubset q_i$, then $r_j \sqsubset r_i$. Therefore $R_0 = \{r_i\}$ is a sequence in $\sqsubseteq$, such that for each i, $q_i \ll r_i$. Since $\ll$ is continuous, $\mathrm{lub}(Q_0) \ll \mathrm{lub}(R_0)$. Finally, by definition of closure, $\mathrm{lub}(R_0) \in \overline{R}$. ∎

We can now show that the substitution property holds for recursively defined functions that do *not* use angelic choice. We treat the case without angelic choice first because this treatment parallels our semantics for recursive function definitions.

Lemma C.2.5. Let

$$\begin{aligned}&\texttt{fun } \mathtt{f}_1(\vec{\mathtt{x}_1} : \vec{\mathtt{S}_1}) : \mathtt{T}_1 = E_1;\\ &\vdots\\ &\texttt{fun } \mathtt{f}_m(\vec{\mathtt{x}_m} : \vec{\mathtt{S}_m}) : \mathtt{T}_m = E_m\end{aligned}$$

be a mutually recursive system of NOAL function definitions, where ∇ does not occur in the E_j. Let Σ be a signature. Let A and B be Σ-algebras such that the carrier set of each non-visible type is flat.

Suppose $\mathcal{R}$ is a simulation relation between A and B for *NomSig* and $\leq$. Then for each j from 1 to m,

$$\mathcal{F}(A)[\![\mathtt{f}_j]\!]\ \mathcal{R}_{\vec{\mathtt{S}_j}\to\mathtt{T}_j}\ \mathcal{F}(B)[\![\mathtt{f}_j]\!]. \tag{C.12}$$

Proof: For each j, let $\mathtt{G}_j$ be the function abstract of the form

$$\texttt{fun}(\vec{\mathtt{x}_j} : \vec{\mathtt{S}_j}) : \mathtt{T}_j = \texttt{bottom}[\mathtt{T}_j].$$

Let $k \in \{1, \ldots, m\}$ be given. Let $\vec{q}$ and $\vec{r}$ have the same length as $\vec{\mathtt{x}_k}$ and be such that $\vec{q}\,\mathcal{R}_{\vec{\mathtt{S}_k}}\,\vec{r}$. Let $\eta_1(\vec{\mathtt{x}_k}) = \vec{q}$ and $\eta_2(\vec{\mathtt{x}_k}) = \vec{r}$. By construction, $\eta_1\ \mathcal{R}\ \eta_2$.

Let $D_{(j,i)}$ be a choice family, and let $Q_k(\vec{q}) = \langle\hat{q}_i\rangle$ be a sequence in $\sqsubseteq$ such that for each i, $\hat{q}_i$ is a possible result of $\mathcal{M}[\![D_{(k,i)}[\vec{\mathtt{G}}/\vec{\mathtt{f}}]]\!](A, \eta_1)$. Let $R_k(\vec{r}) = \langle\hat{r}_i\rangle$, be a sequence in $\sqsubseteq$, where for each i, $\hat{r}_i$ is a possible result of $\mathcal{M}[\![D_{(k,i)}[\vec{\mathtt{G}}/\vec{\mathtt{f}}]]\!](B, \eta_2)$ and $\hat{q}_i\ \mathcal{R}_{\mathtt{T}_k}\ \hat{r}_i$. Such a sequence can be found, because $D_{(k,i)}[\vec{\mathtt{G}}/\vec{\mathtt{f}}]$ is recursion-free, (thus Lemma 6.2.1 applies) and because $\mathcal{R}$ is strongly monotonic. Since $\mathcal{R}_{\mathtt{T}_k}$ is a continuous relation, $\mathrm{lub}(Q_k(\vec{q}))\ \mathcal{R}_{\mathtt{T}_j}\ \mathrm{lub}(R_k(\vec{r}))$.

Let $DD_k(\vec{q})$ denote the set of all such sequences $Q_k(\vec{q})$ in $\sqsubseteq$ for all choice families and let $DD_k(\vec{r})$ be similarly defined. By the above, for every $Q_k(\vec{q}) \in DD_k(\vec{q})$, there is some $R_k(\vec{r}) \in DD_k(\vec{r})$ (obtained using the same choice family) such that $\mathrm{lub}(Q_k(\vec{q}))\ \mathcal{R}_{\mathtt{T}_k}\ \mathrm{lub}(R_k(\vec{r}))$. Therefore, we have

$$\{\mathrm{lub}(Q_k(\vec{q})) \mid Q_k(\vec{q}) \in DD_k(\vec{q})\}\ \mathcal{R}_{\mathtt{T}_k}\ \{\mathrm{lub}(R_k(\vec{r})) \mid R_k(\vec{r}) \in DD_k(\vec{r})\}. \tag{C.13}$$

Since these sets of least upper bounds are related by $\mathcal{R}_{\mathtt{T}_k}$ and $\mathcal{R}_{\mathtt{T}_k}$ is strongly monotonic and continuous, by Lemma C.2.4 the closures of these sets are related by $\mathcal{R}_{\mathtt{T}_k}$.

Therefore,

$$\begin{aligned} \mathcal{F}(A)[\![\mathtt{f}_k]\!](\vec{q}) &\overset{\text{def}}{=} \overline{\{\text{lub}(Q_k(\vec{q})) \mid Q_k(\vec{q}) \in DD_k(\vec{q})\}} & \text{(C.14)} \\ \mathcal{R}_{\mathtt{T}_k} & \overline{\{\text{lub}(R_k(\vec{r})) \mid R_k(\vec{r}) \in DD_k(\vec{r})\}} & \text{(C.15)} \\ &\overset{\text{def}}{=} \mathcal{F}(B)[\![\mathtt{f}_k]\!](\vec{r}) & \text{(C.16)} \end{aligned}$$

So for each j from 1 to m,

$$\mathcal{F}(A)[\![\mathtt{f}_j]\!]\ \mathcal{R}_{\vec{\mathtt{S}}_j \to \mathtt{T}_j}\ \mathcal{F}(B)[\![\mathtt{f}_j]\!].$$

∎

We must now deal with recursively defined functions that use angelic choice. Since the semantics of such systems is given by first replacing angelic choice with erratic choice, we use the following lemma to show how the set of possible results of an expression is affected by this substitution.

Lemma C.2.6. Let A be an algebra. Let X be a set of typed identifiers. Let $\eta \in ENV(X, A)$ be an environment such that for each function identifier $\mathtt{f}$, $\eta(\mathtt{f})$ is monotonic. Let γ be a NOAL expression.

Suppose γ' is derived from γ by replacing all the angelic choice ($\bigtriangledown$) operators in γ with erratic choice operators ($[\!]$). Then

$$\mathcal{M}[\![\gamma']\!](A, \eta) \sqsubseteq_E \mathcal{M}[\![\gamma]\!](A, \eta). \tag{C.17}$$

Proof: (by induction on the structure of NOAL expressions.)

As a basis, if γ is an identifier, `bottom[T]` for some type `T`, or the invocation of a nullary generic operation symbol, then the result is trivial.

For the inductive step, suppose that the result holds for each subexpression. As Broy points out [Bro86, Theorem 3.2], the meaning of each expression except angelic choice that has subexpressions $\gamma_1, \ldots, \gamma_n$ has the form

$$\mathcal{M}[\![\text{expr}(\vec{\gamma})]\!](A, \eta) = \bigcup_{\vec{q} \in \mathcal{M}[\![\vec{\gamma}]\!](A, \eta)} h(\vec{q})$$

for some monotonic set-valued function h. In particular, we have assumed that each operation of an algebra is monotonic and by hypothesis, for each function identifier $\mathtt{f}$,

$\eta(\mathtt{f})$ is monotonic in $\sqsubseteq_E$. If expr($\vec{\gamma'}$) is derived from expr($\vec{\gamma}$) by replacing all occurrences of ∇ with $[\!]$, then by the inductive hypothesis we have $\mathcal{M}[\![\vec{\gamma'}]\!](A,\eta) \sqsubseteq_E \mathcal{M}[\![\vec{\gamma}]\!](A,\eta)$. Therefore,

$$\begin{aligned}
\mathcal{M}[\![\mathrm{expr}(\vec{\gamma'})]\!](A,\eta) &= \bigcup_{\vec{q}\in\mathcal{M}[\![\vec{\gamma'}]\!](A,\eta)} h(\vec{q}) && \text{(C.18)}\\
&\sqsubseteq_E \bigcup_{\vec{q}\in\mathcal{M}[\![\vec{\gamma}]\!](A,\eta)} h(\vec{q}) && \text{(C.19)}\\
&= \mathcal{M}[\![\mathrm{expr}(\vec{\gamma})]\!](A,\eta). && \text{(C.20)}
\end{aligned}$$

Finally, consider the expressions $\gamma_1 \nabla \gamma_2$ and the derived expression $\gamma_1'[\!]\gamma_2'$. By definition of NOAL,

$$\begin{aligned}
\mathcal{M}[\![\gamma_1'[\!]\gamma_2']\!](A,\eta) &\stackrel{\text{def}}{=} \mathcal{M}[\![\gamma_1']\!](A,\eta) \cup \mathcal{M}[\![\gamma_2']\!](A,\eta) && \text{(C.21)}\\
&\sqsubseteq_E \mathcal{M}[\![\gamma_1]\!](A,\eta) \cup \mathcal{M}[\![\gamma_2]\!](A,\eta) && \text{(C.22)}\\
&\stackrel{\text{def}}{=} \mathcal{M}[\![\gamma_1[\!]\gamma_2]\!](A,\eta) && \text{(C.23)}\\
&\sqsubseteq_E \mathcal{M}[\![\gamma_1 \nabla \gamma_2]\!](A,\eta), && \text{(C.24)}
\end{aligned}$$

because $\mathcal{M}[\![\gamma_1[\!]\gamma_2]\!](A,\eta)$ differs from $\mathcal{M}[\![\gamma_1 \nabla \gamma_2]\!](A,\eta)$ in that the former may contain $\perp$ when the latter does not. ∎

The above lemmas allow us to reach our goal for this section, which is the next lemma. This lemma shows that the substitution property holds for systems of recursively defined functions that may use angelic choice. This lemma is the same as Lemma 6.2.2 from Chapter 6.

Lemma C.2.7. Let

$$\begin{aligned}
&\mathbf{fun}\ \mathtt{f}_1(\vec{\mathtt{x}_1} : \vec{\mathtt{S}_1}) : \mathtt{T}_1 = E_1;\\
&\vdots\\
&\mathbf{fun}\ \mathtt{f}_m(\vec{\mathtt{x}_m} : \vec{\mathtt{S}_m}) : \mathtt{T}_m = E_m
\end{aligned}$$

be a mutually recursive system of NOAL function definitions. Let Σ be a signature. Let A and B be Σ-algebras such that the carrier set of each non-visible type is flat.

Suppose $\mathcal{R}$ is a simulation relation between A and B for *NomSig* and $\leq$. Then for each j from 1 to m,

$$\mathcal{F}(A)[\![\mathtt{f}_j]\!]\ \mathcal{R}_{\vec{\mathtt{S}_j}\to\mathtt{T}_j}\ \mathcal{F}(B)[\![\mathtt{f}_j]\!]. \qquad \text{(C.25)}$$

Proof: Let $E_{(j,0)}$ be derived from E_j by replacing all occurrences of the angelic choice operator ($\sqcap$) with the erratic choice operator ($[]$). Let

$$\begin{array}{l} \texttt{fun } \mathbf{g}_{(1,0)}(\vec{\mathbf{x}_1} : \vec{\mathrm{S}_1}) : \mathrm{T}_1 = E_{(1,0)}; \\ \vdots \\ \texttt{fun } \mathbf{g}_{(m,0)}(\vec{\mathbf{x}_m} : \vec{\mathrm{S}_m}) : \mathrm{T}_m = E_{(m,0)} \end{array}$$

By Lemma C.2.5, for each j from 1 to m,

$$\mathcal{F}(A)[\![\mathbf{g}_{(j,0)}]\!] \; \mathcal{R}_{\vec{\mathrm{S}_j} \to \mathrm{T}_j} \; \mathcal{F}(B)[\![\mathbf{g}_{(j,0)}]\!]. \tag{C.26}$$

The discussion of the semantics of recursive systems above inductively defines a family of approximations for the meaning of $\vec{\mathbf{f}}$ in A, $\mathcal{F}(A)[\![\mathbf{g}_{(j,i)}]\!]$, and a corresponding family for the meaning of $\vec{\mathbf{f}}$ in B, $\mathcal{F}(B)[\![\mathbf{g}_{(j,i)}]\!]$. To show the result we first show two properties of these families of approximations.

The first property is that for all j from 1 to m,

$$\mathcal{F}(A)[\![\mathbf{g}_{(j,i)}]\!] \; \mathcal{R}_{\vec{\mathrm{S}_j} \to \mathrm{T}_j} \; \mathcal{F}(B)[\![\mathbf{g}_{(j,i)}]\!]. \tag{C.27}$$

This follows by induction on i, using Lemma 6.2.1 and Lemma C.2.4.

The second property is that for all natural numbers i, for all j from 1 to m, and for all arguments $\vec{q}$ from the algebra B,

$$\mathcal{F}(B)[\![\mathbf{g}_{(j,i)}]\!](\vec{q}) \;\; \sqsubseteq_E \;\; \mathcal{F}(B)[\![\mathbf{g}_{(j,i+1)}]\!](\vec{q}). \tag{C.28}$$

Intuitively, this should hold because $\mathcal{F}(B)[\![\mathbf{g}_{(j,i+1)}]\!]$ uses angelic choice for deeper recursions than $\mathcal{F}(B)[\![\mathbf{g}_{(j,i)}]\!]$.

The second property is proved by induction on i. For the basis, let j be fixed and let $\vec{q}$ be given. Let η be an environment such that $\eta(\vec{\mathbf{x}_j}) = \vec{q}$ and for all $k \in \{1, \ldots, m\}$, $\eta(\mathbf{f}_k) = \mathcal{F}(B)[\![\mathbf{g}_{(k,0)}]\!]$. Each $\eta(\mathbf{f}_k)$ is monotonic, because the bodies of the $\mathbf{g}_{(k,0)}$ do not use angelic choice. By construction of the $\mathcal{F}(B)[\![\mathbf{g}_{(k,0)}]\!]$,

$$\mathcal{F}(B)[\![\mathbf{g}_{(j,0)}]\!](\vec{q}) = \mathcal{M}[\![E_{(j,0)}]\!](B, \eta). \tag{C.29}$$

By Lemma C.2.6,

$$\mathcal{M}[\![E_{(j,0)}]\!](B, \eta) \;\; \sqsubseteq_E \;\; \mathcal{M}[\![E_j]\!](B, \eta) \tag{C.30}$$

since $E_{(j,0)}$ is derived from E_j by replacing all the angelic choice operators with erratic choice operators. Since by construction, the set $\mathcal{M}[\![E_{(j,0)}]\!](B,\eta)$ is closed,

$$\begin{aligned} \overline{\mathcal{M}[\![E_{(j,0)}]\!](B,\eta)} &= \mathcal{M}[\![E_{(j,0)}]\!](B,\eta) && \text{(C.31)} \\ &\sqsubseteq_E \mathcal{M}[\![E_j]\!](B,\eta) && \text{(C.32)} \\ &\sqsubseteq_E \overline{\mathcal{M}[\![E_j]\!](B,\eta)}. && \text{(C.33)} \end{aligned}$$

By definition,

$$\mathcal{F}(B)[\![\mathbf{g}_{(j,1)}]\!](\vec{q}) = \overline{\mathcal{M}[\![E_j]\!](B,\eta)}. \qquad \text{(C.34)}$$

So combining the above,

$$\mathcal{F}(B)[\![\mathbf{g}_{(j,0)}]\!](\vec{q}) \sqsubseteq_E \mathcal{F}(B)[\![\mathbf{g}_{(j,1)}]\!](\vec{q}). \qquad \text{(C.35)}$$

For the inductive step, assume that for all j and all $\vec{q}$,

$$\mathcal{F}(B)[\![\mathbf{g}_{(j,i-1)}]\!](\vec{q}) \sqsubseteq_E \mathcal{F}(B)[\![\mathbf{g}_{(j,i)}]\!](\vec{q}). \qquad \text{(C.36)}$$

Let η_{i-1} be an environment such that for all $k \in \{1,\ldots,m\}$, $\eta(\mathbf{f}_k) = \mathcal{F}(B)[\![\mathbf{g}_{(k,i-1)}]\!]$ and such that $\eta(\vec{\mathbf{x}_j}) = \vec{q}$. Let η_i be an environment such that for all $k \in \{1,\ldots,m\}$, $\eta(\mathbf{f}_k) = \mathcal{F}(B)[\![\mathbf{g}_{(k,i)}]\!]$ and such that $\eta(\vec{\mathbf{x}_j}) = \vec{q}$. By definition,

$$\begin{aligned} \mathcal{F}(B)[\![\mathbf{g}_{(j,i+1)}]\!](\vec{q}) &= \overline{\mathcal{M}[\![E_j]\!](B,\eta_i)} && \text{(C.37)} \\ \mathcal{F}(B)[\![\mathbf{g}_{(j,i)}]\!](\vec{q}) &= \overline{\mathcal{M}[\![E_j]\!](B,\eta_{i-1})}. && \text{(C.38)} \end{aligned}$$

Furthermore, by induction on the structure of NOAL expressions (as in Lemma C.2.6), the induction hypothesis can be used to show that

$$\mathcal{M}[\![E_j]\!](B,\eta_{i-1}) \sqsubseteq_E \mathcal{M}[\![E_j]\!](B,\eta_i) \qquad \text{(C.39)}$$

So the second property (Formula C.28) holds.

We now turn to the proof of the main result. Let j be fixed and suppose $\vec{q}_j \;\mathcal{R}_{\vec{\mathbf{S}}_j}\; \vec{r}_j$. By definition of NOAL.

$$\begin{aligned} \mathcal{F}(A)[\![\mathbf{f}_j]\!](\vec{q}_j) &\overset{\text{def}}{=} \bigcap_i \mathcal{F}(A)[\![\mathbf{g}_{(j,i)}]\!](\vec{q}_j) && \text{(C.40)} \\ \mathcal{F}(B)[\![\mathbf{f}_j]\!](\vec{r}_j) &\overset{\text{def}}{=} \bigcap_i \mathcal{F}(B)[\![\mathbf{g}_{(j,i)}]\!](\vec{r}_j). && \text{(C.41)} \end{aligned}$$

Suppose $q \in \mathcal{F}(A)[\![\mathtt{f}_j]\!](\vec{q}_j)$. Then for all i, $q \in \mathcal{F}(A)[\![\mathtt{g}_{(j,i)}]\!](\vec{q}_j)$. Since for each i, $\mathcal{F}(A)[\![\mathtt{g}_{(j,i)}]\!]\ \mathcal{R}_{\vec{\mathtt{S}}_j \to \mathtt{T}_j}\ \mathcal{F}(B)[\![\mathtt{g}_{(j,i)}]\!]$, for each i, there is some $r_i \in \mathcal{F}(B)[\![\mathtt{g}_{(j,0)}]\!](\vec{r}_j)$ such that $q\ \mathcal{R}_{\mathtt{T}_j}\ r_i$.

- If $q = \bot$, then since $\mathcal{R}_{\mathtt{T}_j}$ is bistrict, q can only be related to $\bot$. So each r_i is $\bot$, and thus $\bot \in \mathcal{F}(B)[\![\mathtt{f}_j]\!](\vec{r}_j)$. Since q is related to some element of $\mathcal{F}(B)[\![\mathtt{f}_j]\!](\vec{r}_j)$, it must be that $\mathcal{F}(A)[\![\mathtt{f}_j]\!](\vec{q}_j)\ \mathcal{R}_{\mathtt{T}_j}\ \mathcal{F}(B)[\![\mathtt{f}_j]\!](\vec{r}_j)$.

- If q is a proper instance of a visible type, then since $\mathcal{R}$ is V-identical, each $r_i = q$, and thus $q \in \mathcal{F}(B)[\![\mathtt{f}_j]\!](\vec{r}_j)$. So $\mathcal{F}(A)[\![\mathtt{f}_j]\!](\vec{q}_j)\ \mathcal{R}_{\mathtt{T}_j}\ \mathcal{F}(B)[\![\mathtt{f}_j]\!](\vec{r}_j)$.

- If q is a proper instance of some non-visible type, then each of the r_i must be instances of a non-visible type as well, since $\mathcal{R}$ is V-identical. Suppose $r_0 \notin \mathcal{F}(B)[\![\mathtt{g}_{(j,1)}]\!](\vec{r}_j)$, then $r_0 \neq r_1$ and hence $r_0 \sqsubset r_1$, since $\mathcal{F}(B)[\![\mathtt{g}_{(j,0)}]\!](\vec{r}_j) \sqsubseteq_E \mathcal{F}(B)[\![\mathtt{g}_{(j,1)}]\!](\vec{r}_j)$. But since r_0 and r_1 are elements of a non-visible type, they are elements of a flat domain, and therefore $r_0 = \bot$. But this contradicts our assumption that q is proper, since $\mathcal{R}$ is bistrict. So it must be that $r_0 = r_1$. By induction on i, it follows that for all natural numbers i, $r_i = r_0$. Therefore $r_0 \in \mathcal{F}(B)[\![\mathtt{f}_j]\!](\vec{r}_j)$. So $\mathcal{F}(A)[\![\mathtt{f}_j]\!](\vec{q}_j)\ \mathcal{R}_{\mathtt{T}_j}\ \mathcal{F}(B)[\![\mathtt{f}_j]\!](\vec{r}_j)$.

So whenever $\vec{q}_j\ \mathcal{R}_{\vec{\mathtt{S}}_j}\ \vec{r}_j$,

$$\mathcal{F}(A)[\![\mathtt{f}_j]\!](\vec{q}_j)\ \mathcal{R}_{\mathtt{T}_j}\ \mathcal{F}(B)[\![\mathtt{f}_j]\!](\vec{r}_j). \tag{C.42}$$

Therefore for all j, by definition we have

$$\mathcal{F}(A)[\![\mathtt{f}_j]\!]\ \mathcal{R}_{\vec{\mathtt{S}}_j \to \mathtt{T}_j}\ \mathcal{F}(B)[\![\mathtt{f}_j]\!]. \tag{C.43}$$

∎

References

[Ada83] *Reference Manual for the Ada Programming Language*. American National Standards Institute, February 1983. ANSI/MIL-STD 1815A.

[BDMN73] Graham M. Birtwistle, Ole-Johan Dahl, Bjorn Myhrhaug, and Kristen Nygaard. *SIMULA Begin*. Auberach Publishers, Philadelphia, Penn., 1973.

[BHJL86] Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy. Object Structure in the Emerald System. *ACM SIGPLAN Notices*, 21(11):78–86, November 1986. OOPSLA '86 Conference Proceedings, Norman Meyrowitz (editor), September 1986, Portland, Oregon.

[Bro86] Manfred Broy. A Theory for Nondeterminism, Parallelism, Communication, and Concurrency. *Theoretical Computer Science*, 45(1):1–61, 1986.

[BW87] Kim B. Bruce and Peter Wegner. Algebraic and Lambda Calculus Models of Subtype and Inheritance (Extended Abstract). 1987. Obtained via Peter Wegner.

[Car84] Luca Cardelli. A Semantics of Multiple Inheritance. In D. B. MacQueen G. Kahn and G. Plotkin, editors, *Semantics of Data Types: International Symposium, Sophia-Antipolis, France*, pages 51–66, Springer-Verlag, New York, N.Y., June 1984. A revised version of this will appear in Information and Control.

[CW85] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction and Polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985. An earlier version was a Brown University TR.

[DMN70] Ole-Johan Dahl, B. Myhraug, and K. Nygaard. *The Simula 67 common base language*. Publication S-22, Norwegian Computing Center, Oslo, Norway, 1970.

[DT88] Scott Danforth and Chris Tomlinson. Type Theories and Object-Oriented Programming. *ACM Computing Surveys*, 20(1):29–72, March 1988.

[EM85] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics. EATCS Monographs on Theoretical Computer Science*, Springer-Verlag, New York, N.Y., 1985.

[GH86a] J. V. Guttag and J. J. Horning. A Larch Shared Language Handbook. *Science of Computer Programming*, 6:135–157, 1986.

[GH86b] J. V. Guttag and J. J. Horning. Report on the Larch Shared Language. *Science of Computer Programming*, 6:103–134, 1986.

[GHW85] J. V. Guttag, J. J. Horning, and J. M. Wing. *Larch in Five Easy Pieces.* Technical Report 5, Digital Systems Research Center, July 1985.

[GMW79] Michael J. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF.* Volume 78 of *Lecture Notes in Computer Science*, Springer-Verlag, New York, N.Y., 1979. The second author is listed on the cover as Arthur J. Milner, which is clearly a mistake.

[Goo75] J. B. Goodenough. Exception Handling: Issues and a Proposed Notation. *Communications of the ACM*, 18(12):683–696, December 1975.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80, The Language and its Implementation.* Addison-Wesley Publishing Co., Reading, Mass., 1983.

[Gra79] George Grätzer. *Universal Algebra.* Springer-Verlag, New York, N.Y., second edition, 1979.

[Hes88] Wim H. Heselink. A Mathematical Approach to Nondeterminism in Data Types. *TOPLAS*, 10(1):87–117, January 1988.

[Hoa69] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–583, October 1969.

[Hoa72] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972.

[JL76] Anita K. Jones and Barbara H. Liskov. A Language Extension for Controlling Access to Shared Data. *IEEE Transactions on Software Engineering*, SE-2(4):277–285, December 1976.

[JL78] Anita K. Jones and Barbara H. Liskov. A Language Extension for Expressing Constraints on Data Access. *Communications of the ACM*, 21(5):358–367, May 1978.

[JM88] Lalita A. Jategaonkar and John C. Mitchell. ML with Extended Patterm Matching and Subtypes (preliminary version). In *ACM Conference on LISP and Functional Programming, Snowbird, Utah*, pages 198–211, 1988.

[LAB*81] Barbara Liskov, Russell Atkinson, Toby Bloom, Eliot Moss, J. Craig Schaffert, Robert Scheifler, and Alan Snyder. *CLU Reference Manual.* Volume 114 of *Lecture Notes in Computer Science*, Springer-Verlag, New York, N.Y., 1981.

[LDH*87] Barbara Liskov, Mark Day, Maurice Herlihy, Paul Johnson, Gary Leavens, Robert Scheifler, and William Weihl. *Argus Reference Manual.* Technical Report 400, Massachusetts Institute of Technology, Laboratory for Computer Science, October 1987. An earlier version appeared as Programming Methodology Group Memo 54 in March 1987.

[LG86] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development.* The MIT Press, Cambridge, Mass., 1986.

[Lis88] Barbara Liskov. Data Abstraction and Hierarchy. *ACM SIGPLAN Notices,* 23(5):17–34, May 1988. Revised version of the keynote address given at OOPSLA '87.

[LL85] Gary T. Leavens and Barbara Liskov. *The Name Clash Problem and a Proposed Solution.* DSG Note 130, Massachusetts Institute of Technology, Laboratory for Computer Science, October 1985.

[LS79] Barbara H. Liskov and Alan Snyder. Exception Handling in CLU. *IEEE Transactions on Software Engineering,* SE-5(6):546–558, November 1979.

[Mit86] John C. Mitchell. Representation Independence and Data Abstraction (preliminary version). In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida,* pages 263–276, ACM, January 1986.

[Nip86] Tobias Nipkow. Non-deterministic Data Types: Models and Implementations. *Acta Informatica,* 22(16):629–661, March 1986.

[Nip87] Tobias Nipkow. *Behavioural Implementation Concepts for Nondeterministic Data Types.* PhD thesis, University of Manchester, May 1987.

[OBr85] Patrick O'Brien. *Trellis Object-Based Environment: Language Tutorial.* Technical Report DEC-TR-373, Eastern Research Lab, Digital Equipment Corp., Hudson, Mass., November 1985.

[SCB*86] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian, and Carrie Wilpolt. An Introduction to Trellis/Owl. *ACM SIGPLAN Notices,* 21(11):9–16, November 1986. OOPSLA '86 Conference Proceedings, Norman Meyrowitz (editor), September 1986, Portland, Oregon.

[Sch86] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, Inc., Boston, Mass., 1986.

[SCW85] Craig Schaffert, Topher Cooper, and Carrie Wilpolt. *Trellis Object-Based Environment: Language Reference Manual.* Technical Report DEC-TR-372, Eastern Research Lab, Digital Equipment Corp., Hudson, Mass., November 1985.

[Sny86a] Alan Snyder. *CommonObjects: An Overview.* Technical Report STL-86-13, Software Technology Laboratory, Hewlett-Packard Laboratories, Palo Alto, California, June 1986.

[Sny86b] Alan Snyder. Encapsulation and Inheritance in Object-Oriented Programming Languages. *ACM SIGPLAN Notices*, 21(11):38–45, November 1986. OOPSLA '86 Conference Proceedings, Norman Meyrowitz (editor), September 1986, Portland, Oregon.

[ST85] Donald Sannella and Andrzej Tarlecki. *On Observational Equivalence and Algebraic Specification*, pages 308–322. Volume 185 of *Lecture Notes in Computer Science*, Springer-Verlag, New York, N.Y., March 1985.

[Sta85] R. Statman. Logical Relations and the Typed λ-Calculus. *Information and Control*, 65(2/3):85–97, May/June 1985.

[Sym84] Symbolics, Inc. *Lisp Machine Manual.* Cambridge, Mass., March 1984. Eight volumes.

[vWMP*77] A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck, C. H. Koster, M. Sintzoff, C. H. Lindsey, L. G. L. T. Meertens, and R. G. Fisker. Revised Report on the Algorithmic Language ALGOL 68. *ACM SIGPLAN Notices*, 12(5):1–70, 1977. This has also been published by Springer-Verlag, New York, N. Y., and in Acta Informatica, volume 5, pages 1-236 (1975).

[Win83] Jeannette Marie Wing. *A Two-Tiered Approach to Specifying Programs.* Technical Report TR-299, Massachusetts Institute of Technology, Laboratory for Computer Science, 1983.

[Win87] Jeannette M. Wing. Writing Larch Interface Language Specifications. *ACM Transactions on Programming Languages and Systems*, 9(1):1–24, January 1987.

OFFICIAL DISTRIBUTION LIST

Director Information Processing Techniques Office Defense Advanced Research Projects Agency 1400 Wilson Boulevard Arlington, VA 22209	2 copies
Office of Naval Research 800 North Quincy Street Arlington, VA 22217 Attn: Dr. R. Grafton, Code 433	2 copies
Director, Code 2627 Naval Research Laboratory Washington, DC 20375	6 copies
Defense Technical Information Center Cameron Station Alexandria, VA 22314	12 copies
National Science Foundation Office of Computing Activities 1800 G. Street, N.W. Washington, DC 20550 Attn: Program Director	2 copies
Dr. E.B. Royce, Code 38 Head, Research Department Naval Weapons Center China Lake, CA 93555	1 copy